Alfred The Hackathon Butler - Developer Guide

By: `Team Alfred` Since: `Sept 2019` Licence: `MIT`

1. Introduction

Alfred is a desktop application to help Hackathons' Human Resource Managers organise a Hackathon event more efficiently and in an organized fashion. Currently, organising a Hackathon is no mean feat, as it requires coordinating across different stakeholders and managing different sources of data (such as participant and mentor information). Instead of managing this process across different media such as Excel spreadsheets and emails, Alfred allows you to coordinate everything in a single, convenient-to-use platform.

In particular, Alfred is a Command Line Interface (CLI) application packaged with a Graphical User Interface (GUI). This means that users are expected to interact with Alfred mainly through the Command Line, but each command executed will invoke a visual response in Alfred.

Any help on the development of Alfred would be greatly appreciated, and there are a couple of ways for you to do so - Contribute to Alfred’s codebase by expanding its features - Improve test coverage - Propose and implement improvements to current features.

This guide seeks to not only kick-start your journey as a contributor to Alfred by quickly getting you up to speed with how Alfred’s codebase and inner workings function, but also hopes to serve as a useful reference to current contributors in times or confusion or when faced with difficulties.

2. Setting up

Refer to the guide here.

3. Design

Alfred was designed with Object-Oriented Programming (OOP) principles in mind, with a focus on respecting the SOLID principles. This section serves to give a description of the major components in the architecture of Alfred. Subsequent sections will provide a lot more information on the inner workings of individual components and how different components interact with each other for Alfred’s various features.

NOTE | Image here to the updated in the later milestones.

3.1. Architecture

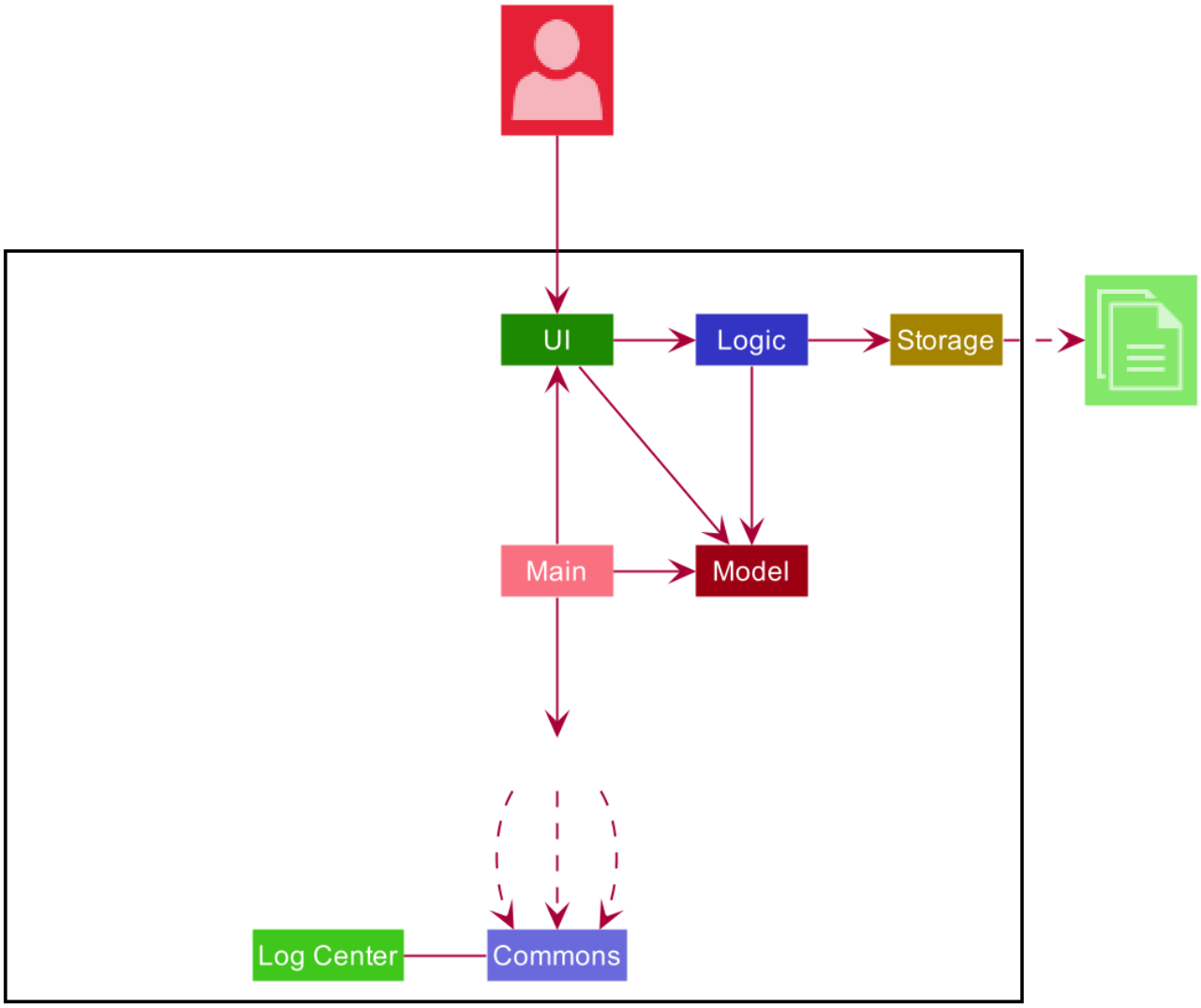


Figure 1. Architecture Diagram

The ***Architecture Diagram*** given above explains the high-level design of the App. Given below is a quick overview of each component.

`Main` has two classes called `Main` and `MainApp`. It is responsible for,

- At app launch: Initializes the components in the correct sequence, and connects them up with each other.
- At shut down: Shuts down the components and invokes cleanup method where necessary.

`Commons` represents a collection of classes used by multiple other components. The following class plays an important role at the architecture level:

`LogsCenter` : Used by many classes to write log messages to the App's log file.

The rest of the App consists of four components.

- `UI`: The UI of the App.
- `Logic`: The command executor.
- `Model`: Holds the data of the App in-memory.
- `Storage`: Reads data from, and writes data to, the hard disk.

Each of the four components

- Defines its *API* in an `interface` with the same name as the Component.
- Exposes its functionality using a `{Component Name}Manager` class.

For example, the `Logic` component (see the class diagram given below) defines it's API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

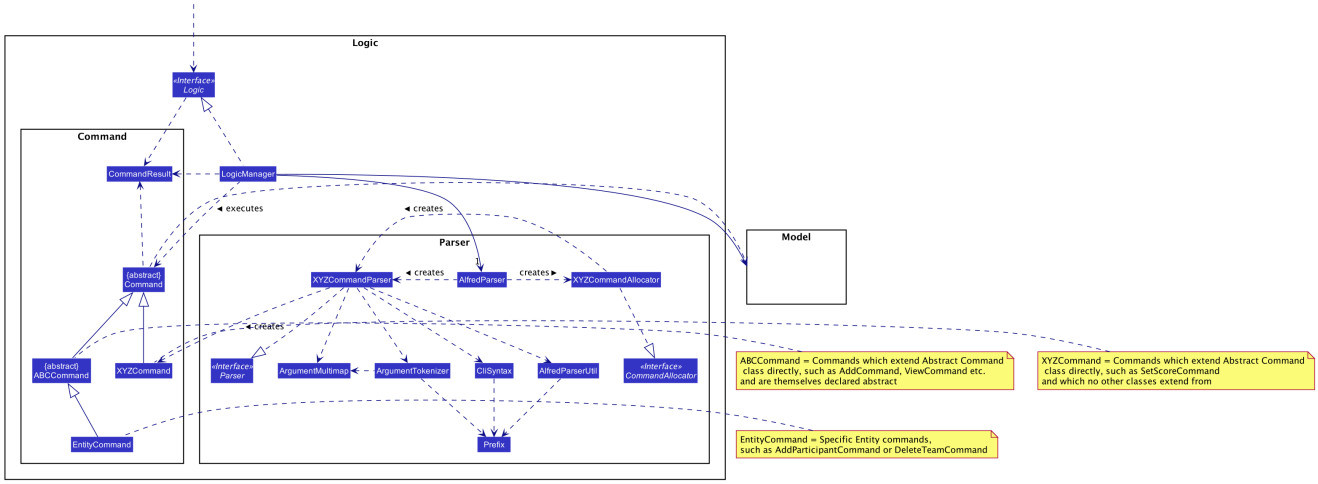


Figure 2. Class Diagram of the Logic Component

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

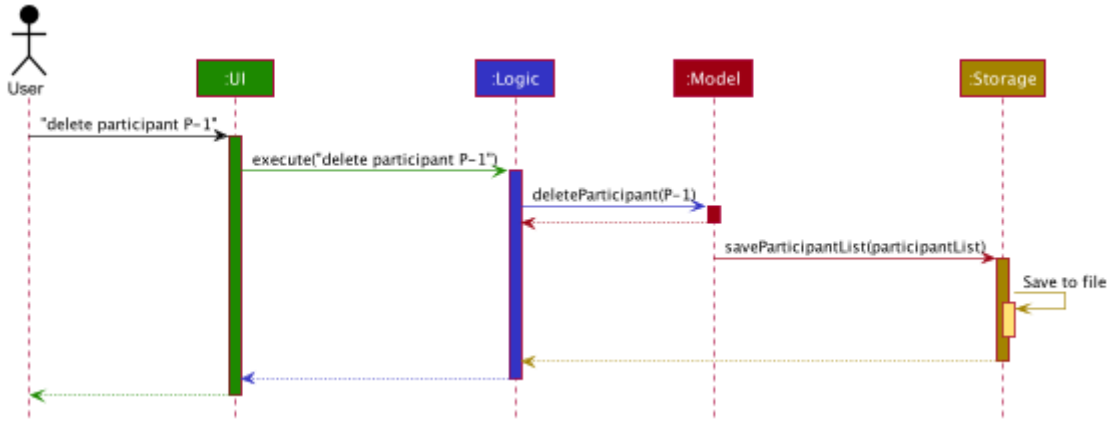


Figure 3. Component interactions for `delete 1` command

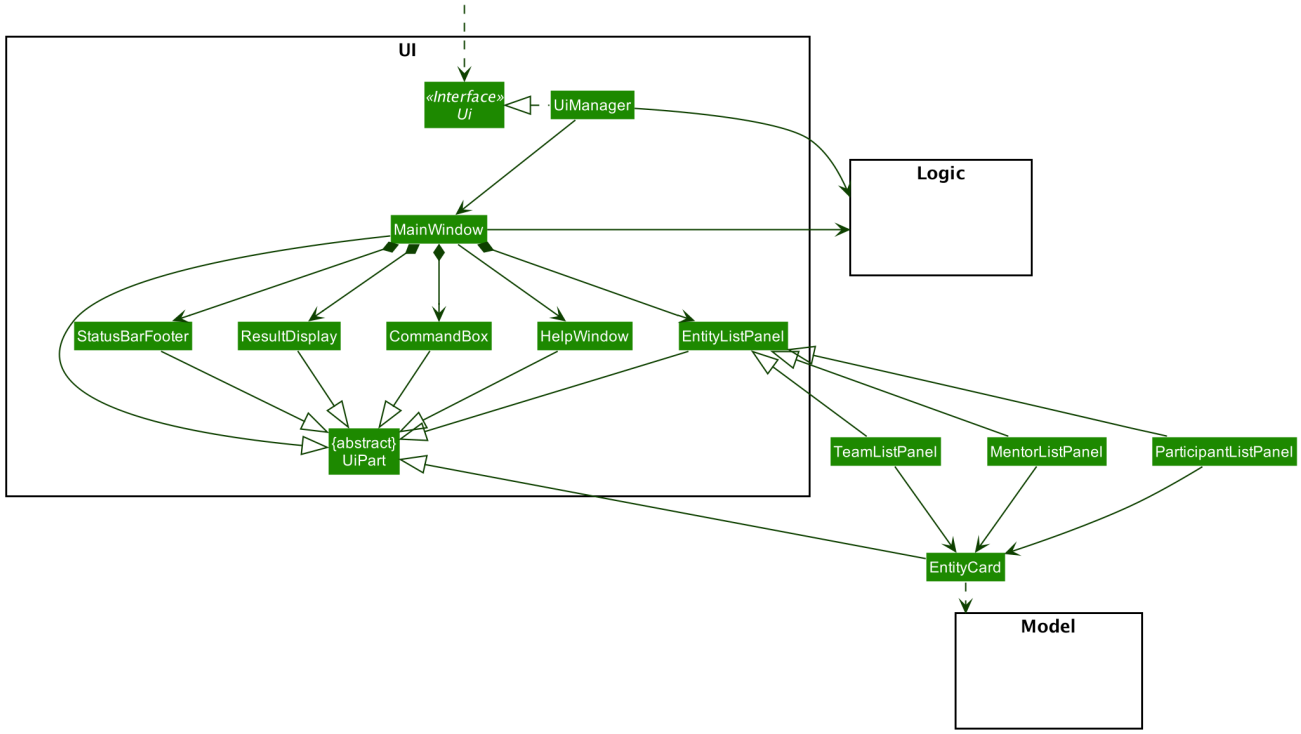
The sections below give more details of each component.

3.2. UI component

NOTE

Structure of the UI Component

UI image and description to be updated with the later milestones.



API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `EntityListPanel`, `StatusBarFooter` 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.
- Looks at the prefix commandType given in the CommandResult(returned after every Command is executed), as displays the specific EntityListPanel respectively.

3.2.1. Design Considerations

1. Ways to update different entity list
    - Alternative 1: The system collects information form Model after each command, to display the entity list as the command result.
        - Pros: Easy to implement with existing Model interface
        - Cons: High degree of dependency between UI components and Model components(high coupling).

        - Cons: Updating of the data is not automatic.
    - Alternative 2: The system uses and Observable interface that observes for changes in the three types of list, namely ParticipantList, TeamList and MentorList.
        - Pros: Low degree of dependency between UI and Model components(low coupling).
        - Pros: The data in GUI is automatically updated.
        - Cons: Harder to implement

Decision: We have decided to go with alternative 2 because low dependency will ensure testability and maintainability of the system.

1. How to generate EntityCard and ListPanel to display different entities
    - Alternative 1: Implement different classes that inherits EntityCard, like TeamCard, ParticipantCard and MentorCard respectively. Additionaly, implement different classes that extents ListPanel, like TeamListPanel and more.
        - Pros: Easy to implement and style respective cards and list panels.
        - Cons: Logic is duplicated many times, one for each type of entity. For example, ParticipantCard and Mentorcard are similar for most fields, except the extra field of Organisation and Specialisation.
        - Cons: Clutters the system with extra classes.
    - Alternative 2: Implement a EntityCard class with a barebone structure. Then dynamically add and morph the fields in Entity card according to the entity type.
        - Pros: No duplication of the same logic and implementation, as abstraction was used.
        - Pros: Lesser class files required.
        - Cons: Harder to implement.

Decision: We have decided to proceed with alternative 2 because this alternative employs that theory of abstraction in programming, and it there is less redundant code in this implementation.

3.3. Logic component

NOTE	Logic Architecture image and description to be updated with the later milestones.

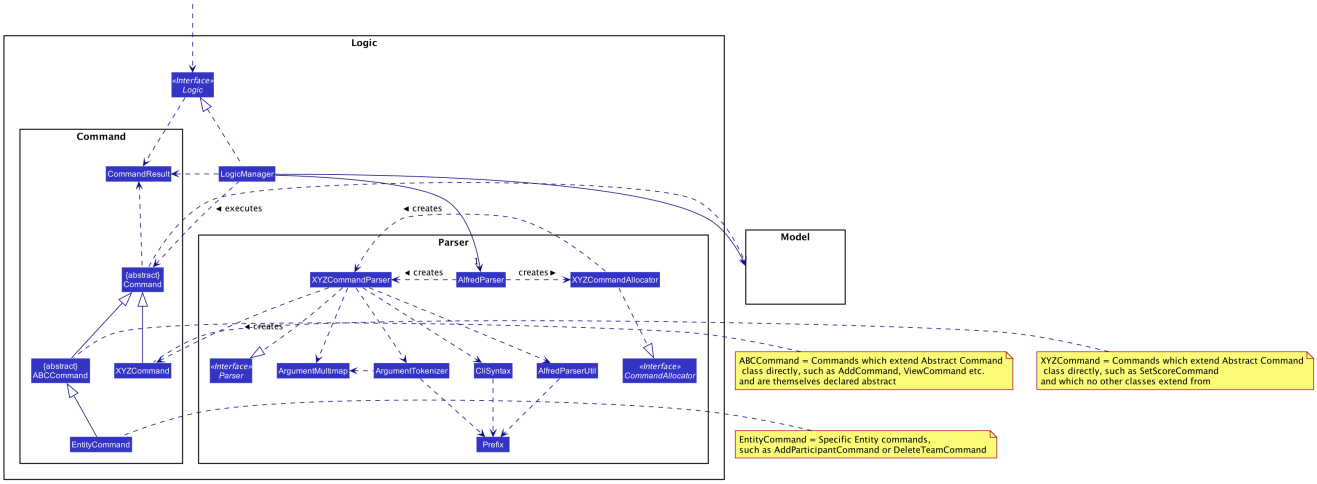


Figure 4. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `AlfredParser` class to parse the user command.
2. This can result in one of two possibilities:
    - a new `CommandAllocator` object is created to allocate the user input to appropriate entity-specific `Parser`. The `CommandAllocator` 's `allocate` method then returns a new Command object which is executed by the LogicManager, or
    - the appropriate `Parser` is directly called if no specifying is required and returns a new `Command` object which is executed by the `LogicManager`.
3. The command execution can affect the `Model` (e.g. adding a participant or deleting a team).
4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.
5. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.

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

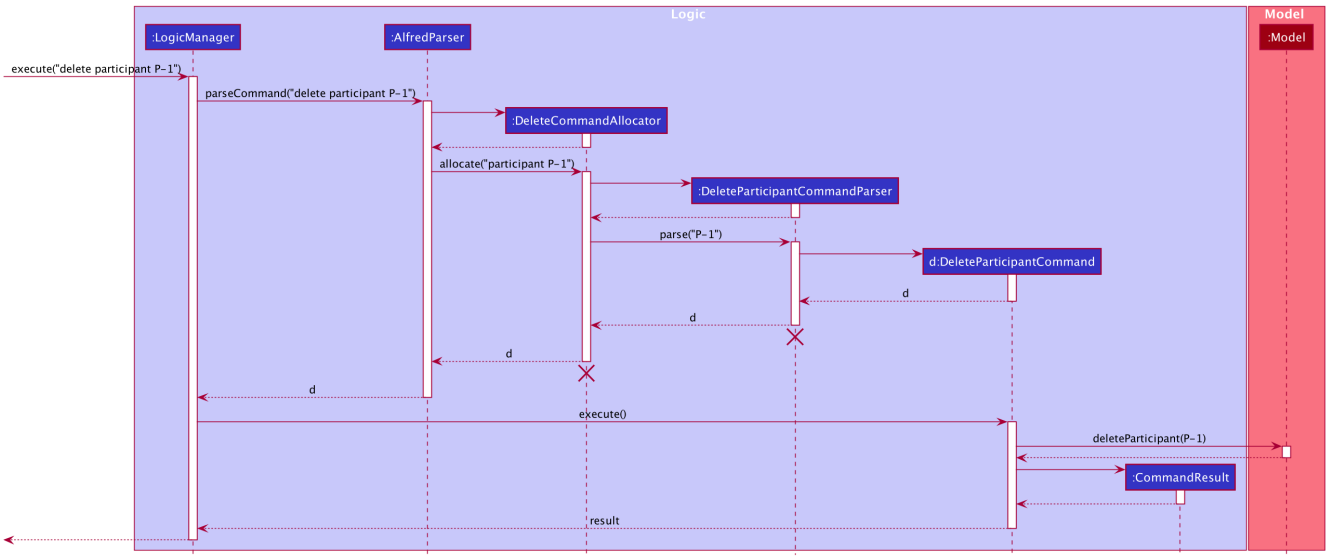


Figure 5. Interactions Inside the Logic Component for the `delete 1` Command

NOTE

The lifeline for `DeleteParticipantCommandParser` and `DeleteCommandAllocator` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

3.4. Model component

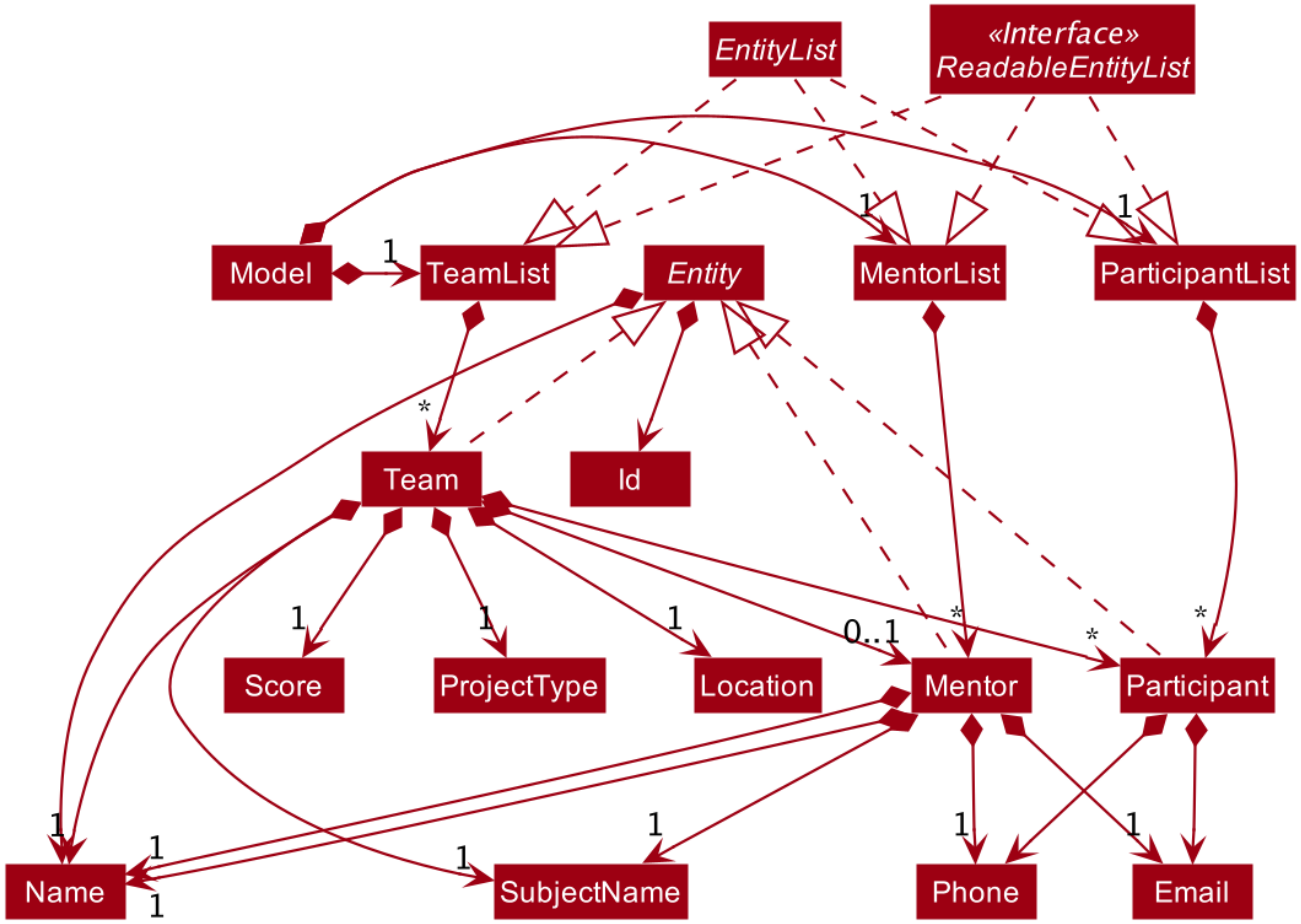


Figure 6. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the lists of our various entities.
- Model is the bridge between Logic and Storage and provides an abstraction of how the data is stored in memory.
- It exposes a `ReadableEntityList` which only has the list method to remind Logic that the data given should not be modified.
- The UI can be bound to these lists so that it automatically updates when the contents of the list change.
- At the heart of the model are observable lists which allow for the dynamic updating of the UI.
- The `Model` interface also serves as an API through which controller can edit the data stored in

memory.

`ModelManager` * ModelManager implements all the methods exposed by the Model Interface. At its heart, it only contains 3 `EntityLists` , 3 `FilteredLists` and a `UserPrefs` Object and does all the validation logic needed for the application

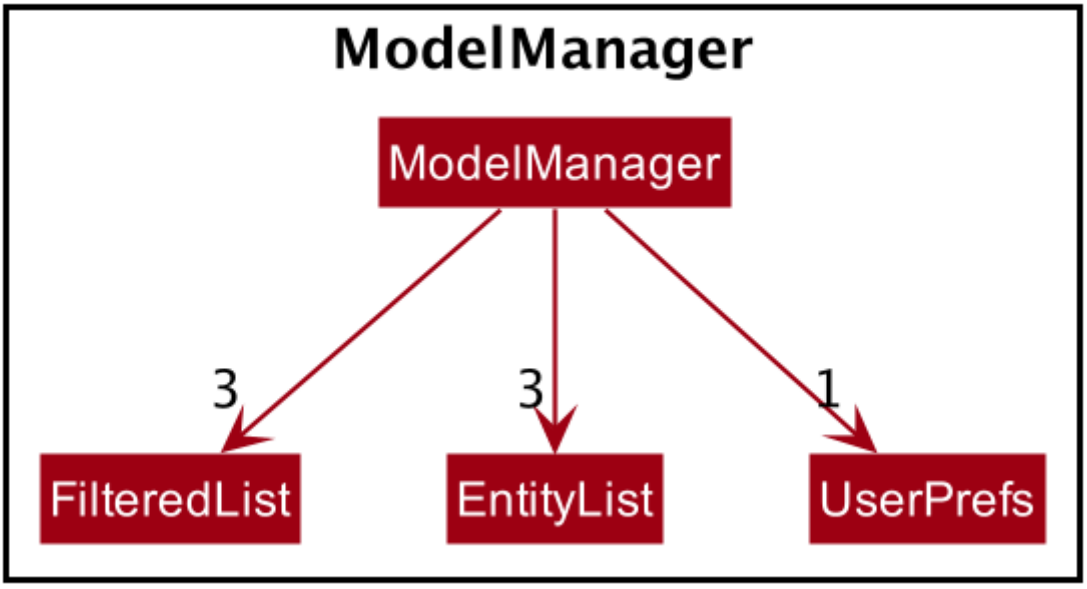


Figure 7. Simple Illustration of ModelManager

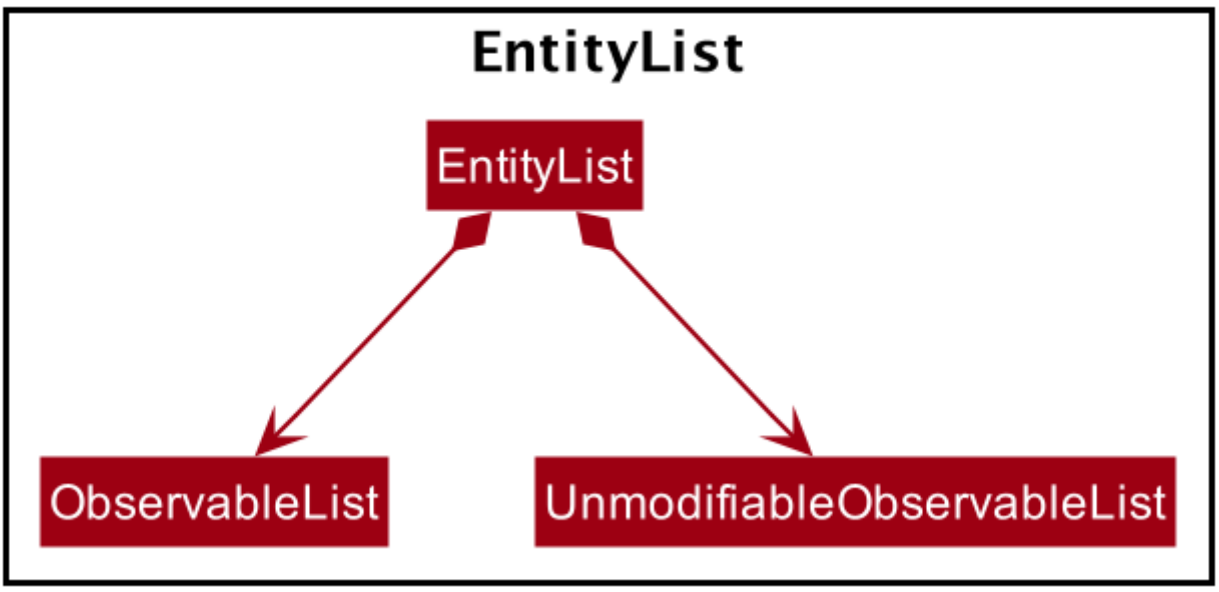


Figure 8. EntityList simplified structure

3.5. Storage component

The Storage component handles the complexities of storing to and reading from disc the Alfred's data. The Storage component transforms the AB3 implementation to support the storage of Alfred's 3 main EntityLists (ParticipantList, MentorList and TeamList) as well as User Preferences. The 3

EntityLists are the main data objects in Alfred, and Storage's purpose is to transform each of the EntityLists into a format that is JSON-Serializable and store the data for each EntityList in separate JSON files. Storage also saves the User Preferences in a JSON file.

NOTE	The Food- and Swag-related features scheduled for release in v2.0, will require some changes in Storage to be made. To be more specific, Storage would need to be updated to support the storage of these other essential data, above and beyond the current support for the storage of the EntityLists.

3.5.1. Purpose and Usage of Storage Component

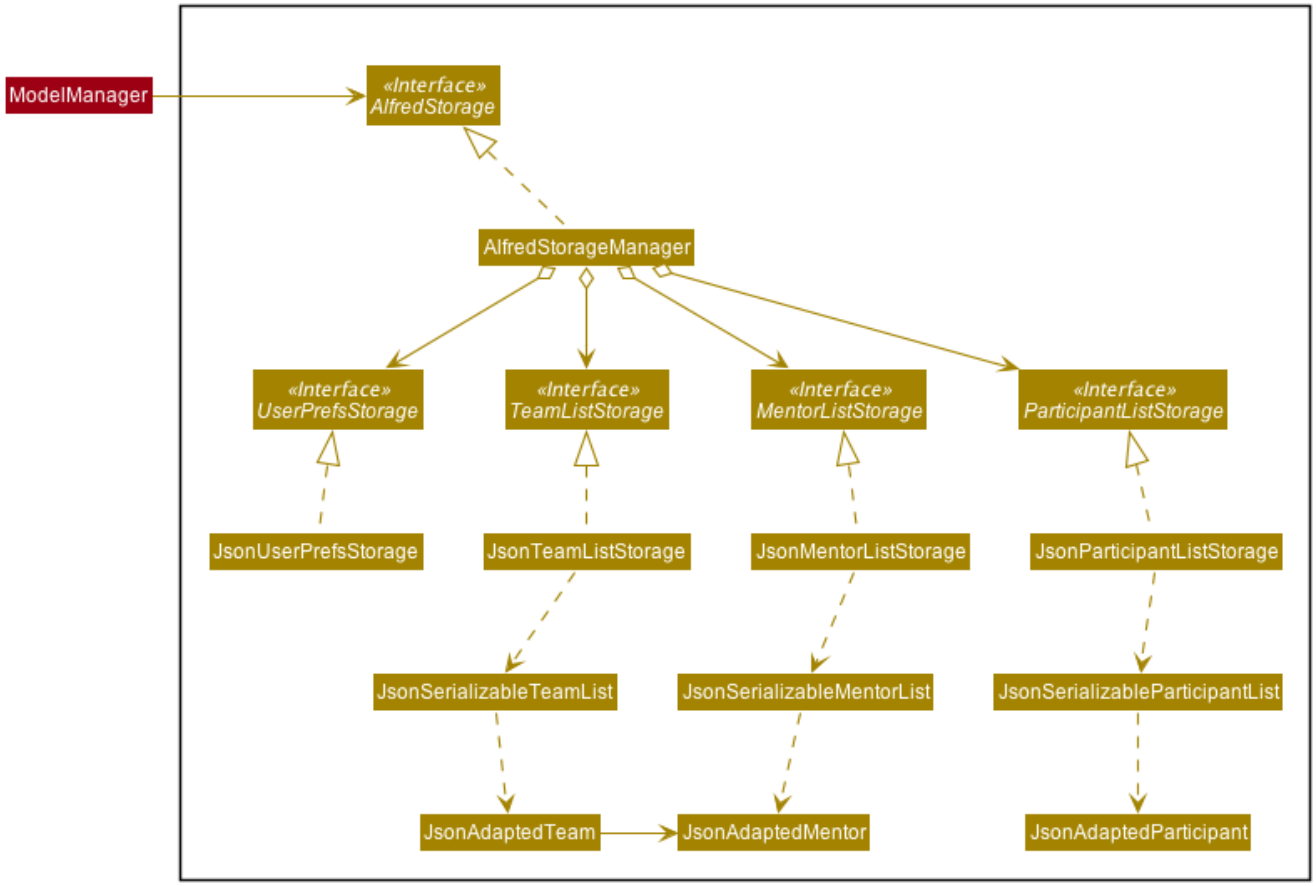


Figure 9. Structure of the Storage Component

API : `AlfredStorage.java`

The `Storage` component saves and reads 4 different ***data types***:

1. `UserPrefs`: User Preferences for Alfred (such as the last used Window Size of the application)
2. `ParticipantList`: Information of all the Participants in Alfred
3. `MentorList`: Information of all the Mentors in Alfred
4. `TeamList`: Information of all the Teams in Alfred

All 4 data types are stored to disc in JSON files. The data is read from the JSON files when Alfred is first start up. It is also important to note that saving is automatic in Alfred. This means that after the execution of each command, the data in Alfred will automatically be saved to disk. This frees the

user from the hassle of constantly calling some form of saving functionality, and ensures that the information in storage is as up-to-date as possible.

NOTE	When reading the JSON files from memory at application start-up, any kind of data corruption in the JSON files will cause Alfred to completely discard the file and re-initialise the data type. If there are missing fields in the JSON file, invalid values in the individual fields or any kind of error while reading the data from the JSON file, Alfred will re-initialise the data type with an empty data type, persisting this newly initialised data type object to disc.

3.5.2. Interacting with the Storage Component

The Storage Component uses the Facade Design Pattern, and exposes the functionality of all the Storage classes to the Model Component solely through the AlfredStorage interface. The exposed functionality was deliberately kept simple, allowing the following methods for each of the 4 data types:

1. `getFilePath()`: Retrieves the location of the JSON file
2. `save()`: saves the data to the JSON file
3. `read()`: reads the data from the JSON file

The Storage component handles the complexities of actually storing to and reading from disc the 4 different data types. As can be seen in the figure at the start of this section, underlying AlfredStorage's simple interface are several classes that ensure the accurate storing and retrieval of Alfred's data from disc. The following are some details of the Storage Component:

- Each EntityList has a designated Storage class (i.e. for Participant, you have `ParticipantListStorage`, `JsonParticipantListStorage` etc.). Hence, for the rest of this explanation, Entity will be used as a generic term for Participants/Mentors/Teams.
- The class implementing the `EntityListStorage` interface is `JsonEntityListStorage`. This class is responsible for providing the read/save functionality for the entire EntityList to `AlfredStorageManager`.
- In order to save the entire `EntityList` in JSON, the class `JsonSerializableEntityList` implements the logic for serialization for the collection of Entities (i.e. EntityList). This is achieved by converting the `EntityList` to a `List<JsonAdaptedEntity>`.
- In order to save each individual Entity, the Entity is in turn converted to a `JsonAdaptedEntity` object. The `JsonAdaptedEntity` class contains the fields of each Entity that are relevant for saving to disc. Hence, this class is directly serialized into JSON, and also has methods for converting the serialized object back into an Entity object for `AlfredStorage` to return to `ModelManager`.
- The conversion of the `JsonAdaptedEntity` object to fields in a JSON file is done by Java's Jackson library.

The following is a concrete example of the storage of a ParticipantList in JSON:

Figure 10. Structure of the JSON File for ParticipantList

The figure above shows the contents of the JSON file storing a ParticipantList containing 2 participants. The data in the red box corresponds to the fields in a single Participant object. These fields are generated by the Jackson library from the serializable `JsonAdaptedParticipant` object, and the entire list of participants in the JSON file is in turn generated because the `JsonSerializableParticipantList` class converts the ParticipantList to a List of `JsonAdaptedParticipant`.

3.5.3. Design Considerations

1. Data Integrity
    - The individual `JsonAdaptedEntity` classes perform validation on every field in the JSON file for each Entity as it attempts to convert the JSON data into an Entity in Alfred. Should the data prove to be invalid, an error is thrown upwards to ModelManager and a new EntityList is initialised in memory, effectively discarding the old EntityList.
    - This design was to ensure that any form of tampering of the code, malicious or inadvertent, will not result in data inconsistencies in Alfred.
    - In future implementations, it would be best to ensure that the JSON file is encrypted and secured to minimise opportunities for tampering with the data.
2. Single Responsibility Principle
    - Each class exists for a very specific purpose. A class is provided for each Entity.
    - Purpose (From Top-Down):
        - Exposing read/save functionality for EntityList: `JsonEntityListStorage`
        - Serializing EntityList: `JsonSerializableEntityList`
        - Serializing one Entity: `JsonAdaptedEntity`
3. Interface Segregation
    - Each interface is kept as minimal as possible and targets a specific Entity type.
    - AlfredStorage is an interface that extends multiple interfaces to expose the read/save functionality required by Model.

4. Dependency Inversion
    - Alfred's Model is dependent on the interface AlfredStorage, and the implementation is provided through AlfredStorageManager. Hence, all functionality provided are first stipulated in the AlfredStorage interface.

3.6. Common classes

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

4. Implementation

To best address the numerous and varying needs of Hackathon organizers, we have packed Alfred with a multitude of different features, each of which seeks to provide the user with the optimal means of tending to their hackathon organising needs. However, with a host of features comes a sizable codebase which can make it daunting for new programmers, and often even veteran Alfred programmers, to understand how certain features have been implemented. This section aims to empower you by introducing you to some of the noteworthy features currently present within Alfred, along with few that hope to add in the future, so that you may better grasp the implementation of these features in a bid to contribute new features and improvements to existing features within Alfred. Despite our best efforts to make the explanations of Alfred's features' as comprehensive as possible, we do advice not to solely rely on this guide for understanding and encourage you to experiment with each feature's code in order to truly grasp it.

4.1. Bulk Registration

The Bulk Registration feature, referred as the import command, allows you to add multiple entities into Alfred at once through a CSV file. The file must be stored locally as Alfred will attempt to retrieve it through the file path provided by the user. In order for the import command to successfully execute, it is required that the CSV file is formatted according to Alfred's requirements, which you can read more about in our user guide.

This feature will be explained further in the following subsections.

4.1.1. Implementation Overview

Since this feature manages data from a CSV file, import command relies on the `CsvUtil` class. The `CsvUtil` class handles reading from and writing data to a CSV file. Below shows the relationships between different classes in Alfred.

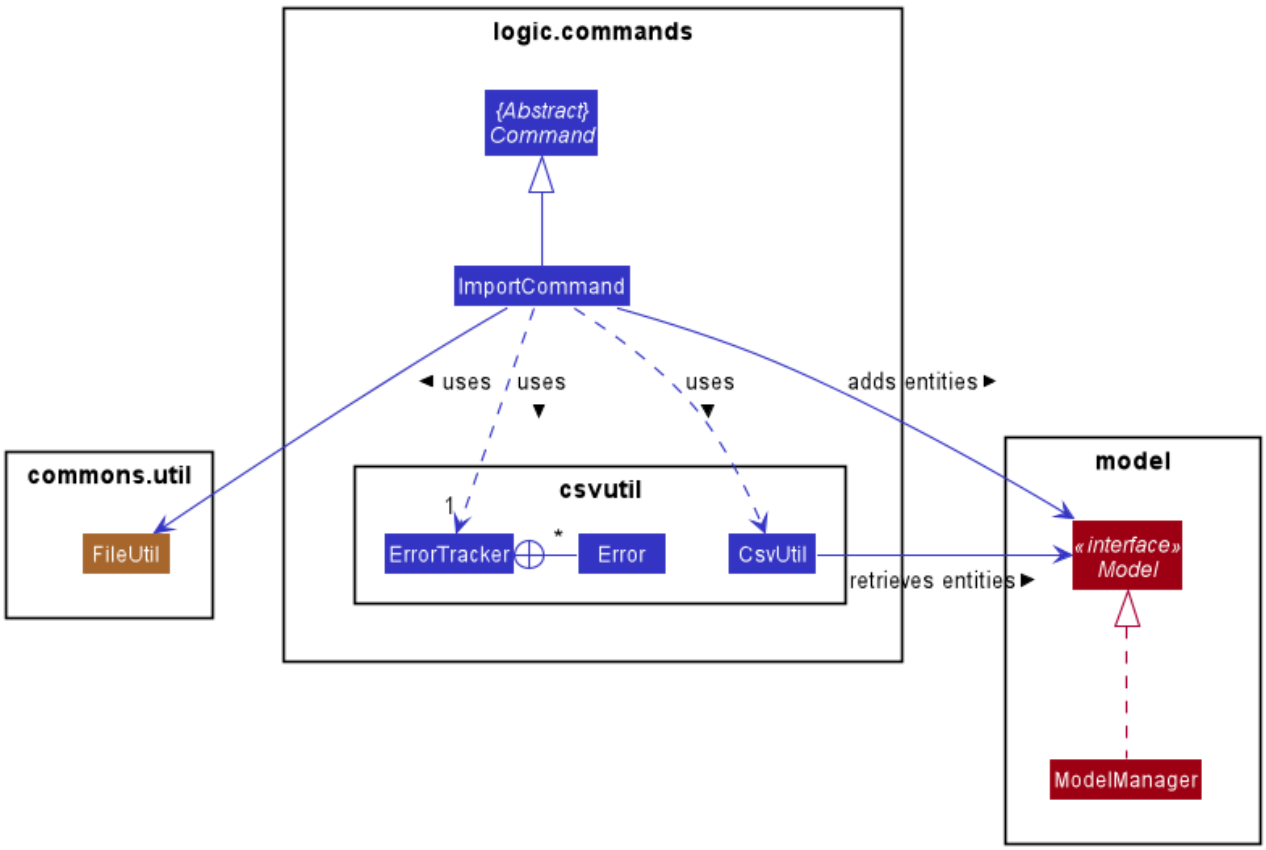


Figure 11. Import Command Class Diagram

In the above class diagram, you can see that

1. The `ImportCommand` uses the `FileUtil` class, and this is so for a number of reasons. First is to validate whether user inputted file path is, in fact, a valid file path. Once it is verified, another check is done to see if the file exists at the given file path. If the file is not able to be located, the `ImportCommand` will not complete its execution.

2. In addition to the `CsvUtil` class, the `ImportCommand` also utilizes an `ErrorTracker` class. This class will store any lines in the CSV file that is invalid along with the reason why it is so. Each `Error` object referenced by the `ErrorTracker` will correspond to one line in the CSV file and the cause of the error.

Now, the sections below will give a detailed explanation of different portions of this feature.

4.1.2. Implementation: `ImportCommand`

Once a valid user input is parsed and passed into the `ImportCommand`, the command will open the file and read its content line by line. Each line is then parsed into the corresponding entity by the `CsvUtil` class. This will be explained further below. The following sequence diagram shows the steps involved in mass importing data into Alfred.

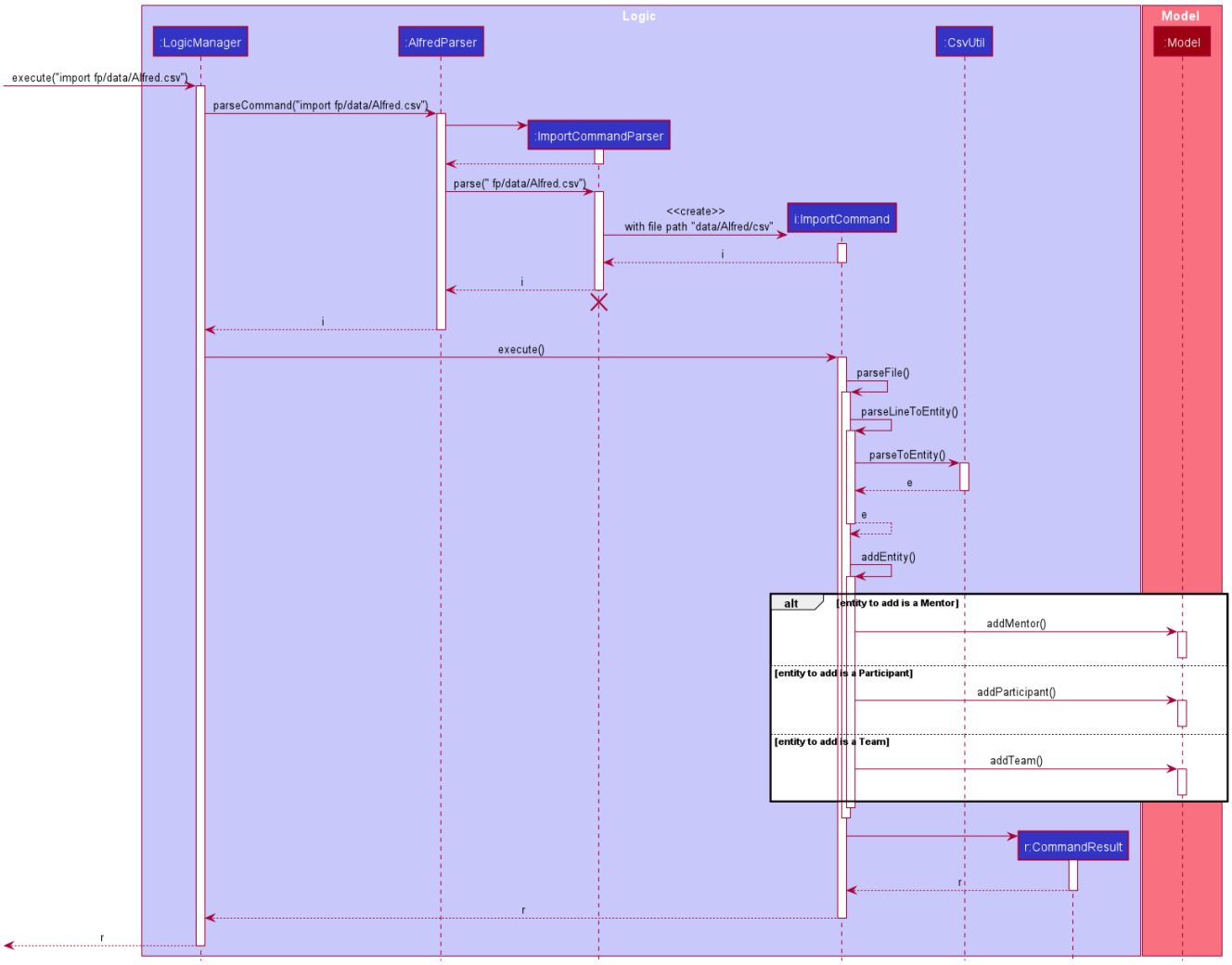


Figure 12. Import Command Sequence Diagram

`FileUtil` shown in the class diagram was omitted from the above sequence diagram for simplicity as it adds little to the overall flow of execution.

As the above figure shows, the file path from the user input is extracted and passed as a field for `ImportCommand`. Then, Alfred proceeds to convert file content into relevant entities.

When `ImportCommand` parses and adds entities to `Model`, it is crucial that teams are the last entities to be added. In the above sequence diagram, this process of buffering teams was also omitted for simplicity. Basically, in the `parseLineToEntity()` method, if a line in CSV file corresponds to a team, the line will be buffered to be parsed after all the other lines have been parsed. The reason for this is because teams may have dependencies on other participants and mentors. It is required that all of the participants and mentors associated with a team, say Team A, exist inside the `Model` before Team A can be added.

So as `ImportCommand` accesses the CSV file line by line, the line representing a team will be stored in a `Queue<String>` for later use. When the end of file is reached and all other participants and mentors are parsed and added to `Model`, the `ImportCommand` will poll from the `Queue`, parse into a relevant team, and add it to the `Model` until the `Queue` is empty.

4.1.3. Implementation: Parsing of Entities by `CsvUtil`

As mentioned before, the `CsvUtil` class is used to aid in parsing of CSV lines into entities. The process in which `CsvUtil` parses each entity is heavily dependent on the different fields each entity has. If you are not familiar with this yet, please check out our user guide.

The process of parsing a line into a mentor or a participant is very similar, so two entities - participant and team - will be explained.

First, the following is a sequence diagram for parsing a CSV line into a participant. The line to be parsed is `"P,ID,Bruce Wayne,12345678,wbruce@wayne.ent"`.

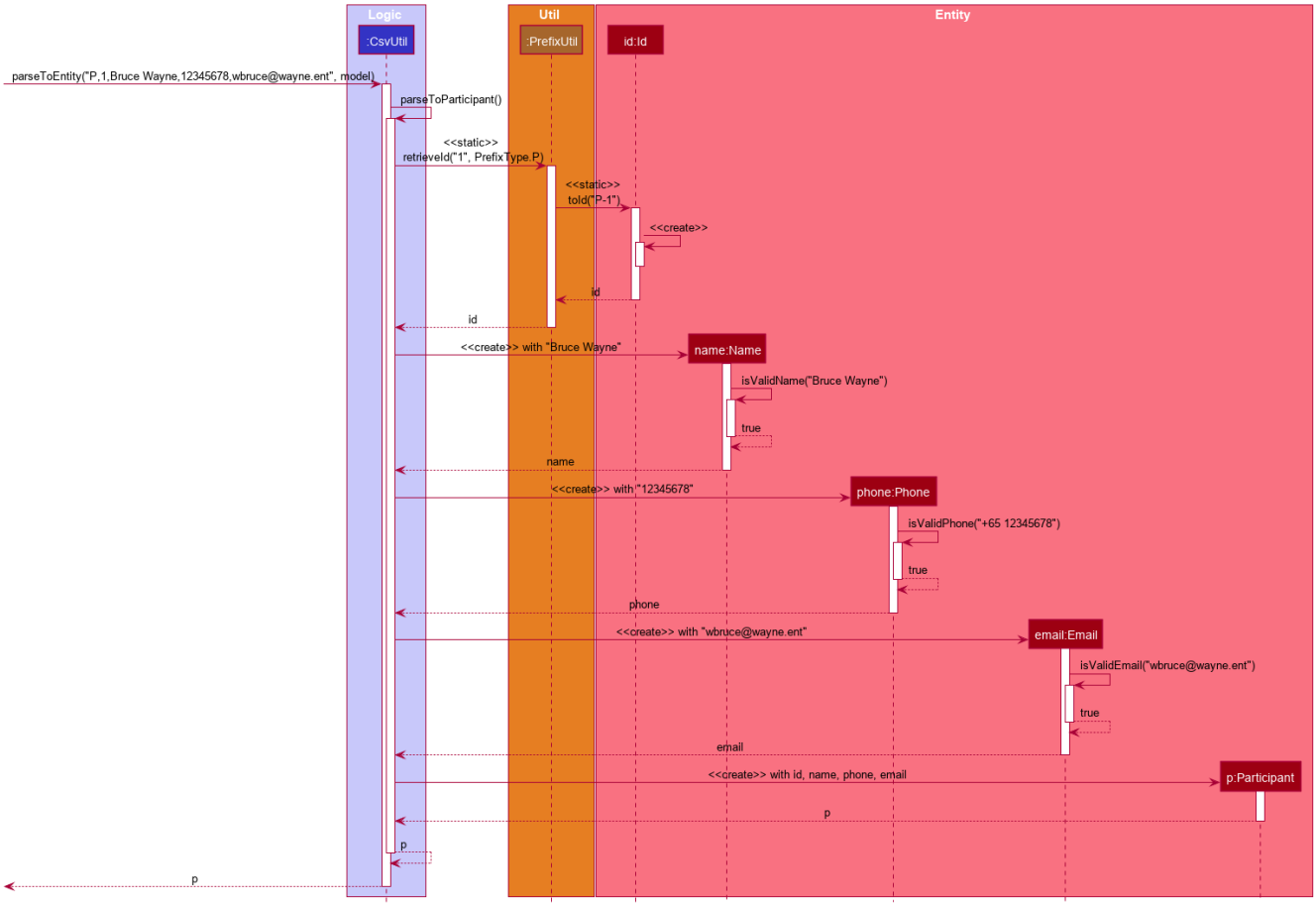


Figure 13. `CsvUtil`: Parsing to Participant Sequence Diagram

Given CSV line is first split by commas (also allows commas surrounded by arbitrary number of spaces). Then, each `String` in the array is (attempted to be) converted into corresponding fields of a Participant. As the diagram shows, each field class has its own method for checking if the given `String` argument is valid - in the form of `isValidField()` method, where `Field` is replaced by its respective class name. Once each field is successfully converted, a Participant is created with the parsed fields. The process of parsing into a Mentor is practically equivalent of that of a Participant. The only change is in the fields being parsed.

Next is a sequence diagram for parsing a CSV line into a team. The line to be parsed is `"T,,Justice League,[P-1|P-2],M-1,Social,100,Save the Earth,1"`.

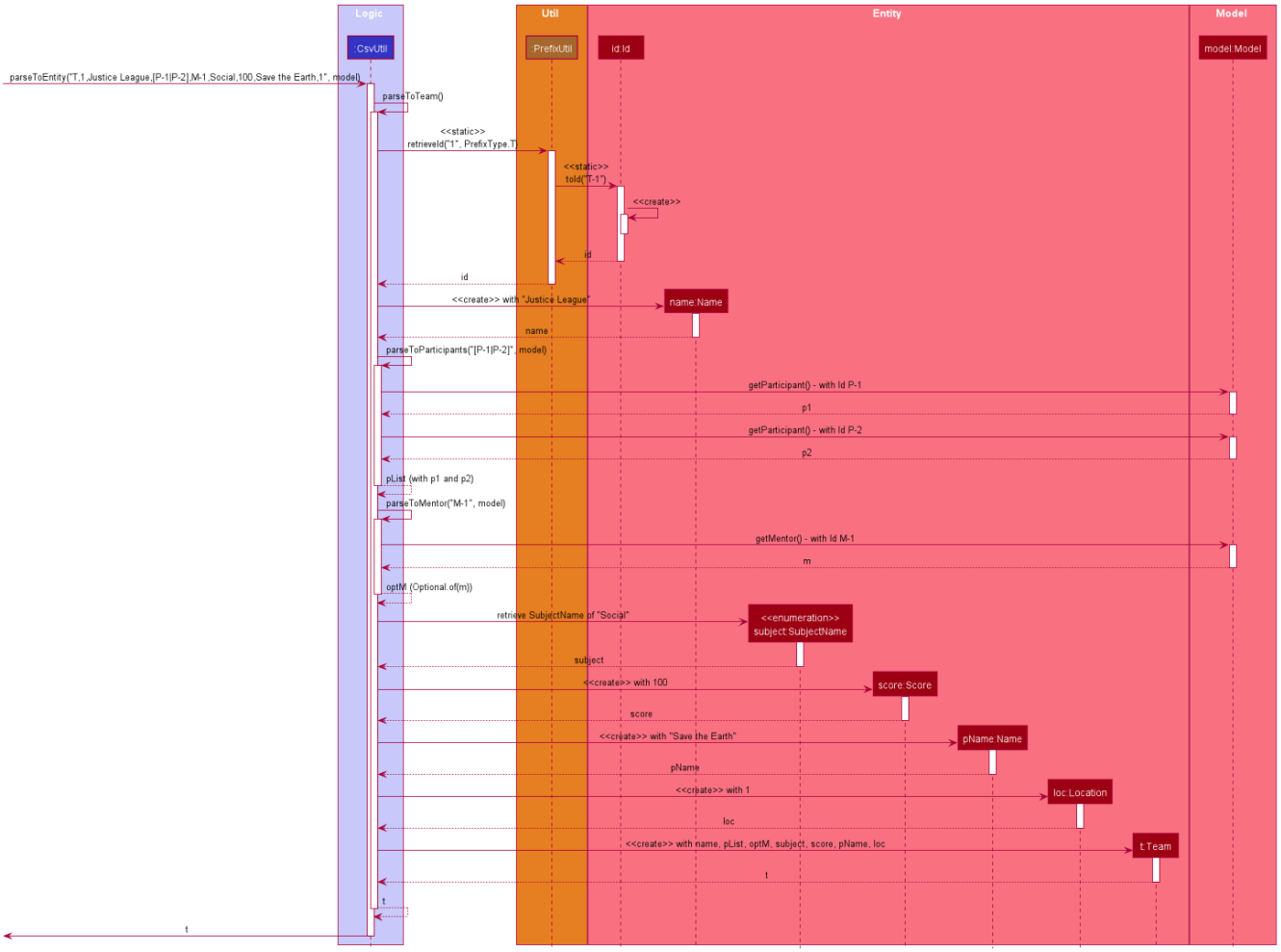


Figure 14. `CsvUtil`: Parsing to Team Sequence Diagram

Also for teams, each corresponding `String` is converted to its field counterpart just like participants and mentors. Hence, `isValidField()` method was omitted from the diagram. The difference lies in the fact that for teams, Alfred must check if any participants or mentors it makes a reference to actually exists in `Model`. Thus, `CsvUtil` calls `getParticipant()` and `getMentor()` methods exposed by the `Model` class. If there are any exceptions raised while retrieving the participants and mentors, that line in CSV will not be loaded onto Alfred.

4.1.4. Implementation: Outputting Error File

4.1.5. Design Considerations

4.2. Export Feature

The export feature will be used to unload the data in Alfred to an external CSV file. This functionality will prove useful when you or the user wishes to share the aggregate data of a particular hackathon event with other people or organization or wishes to keep a record of past hackathons for future references.

4.3. Undo/Redo feature

The Undo/Redo feature, as the name suggests, allows you to undo and redo commands. Only

commands that alter the state of the data in Alfred can be undone/redone. The state of the 3 EntityLists (ParticipantList, MentorList and TeamList) is tracked across the execution of different commands, and the state can be recovered through the use of the undo/redo feature. The last used IDs for each of the 3 EntityLists are also saved.

The feature has been updated in v1.4 to support multiple undos/redos. This means that invoking `undo N`/`redo N` on Alfred, where `N` is an integer, allows you to undo/redo `N` commands at one go.

To undo/redo to next immediate command, simply invoking `undo`/`redo` on Alfred would suffice, as it implicitly calls `undo 1`/`redo 1` in the code.

This feature is a convenience feature as it allows users of Alfred to quickly correct and recover from mistakes, greatly increasing the utility of the application.

NOTE	Only a maximum of 50 data states is stored in `ModelHistoryManager` at any one point in time. The addition of any more data states will result in the discarding of the oldest data state.

4.3.1. Implementation

The general idea is as follows: The undo/redo mechanism is mainly facilitated by the interface `ModelHistory` and its implementation `ModelHistoryManager`. Alfred's data is held in memory within the `ModelManager` object. After the execution of commands that mutate the data in Alfred, a deep copy of all 3 EntityLists is made and saved as a `ModelHistoryRecord` in `ModelHistoryManager`. A deep copy is necessary to ensure that any subsequent changes to data will not alter the data in the `ModelHistoryRecord`, allowing each `ModelHistoryRecord` to serve as a pristine record of the state of the data in Alfred at the end of the execution of each command.

Whenever the `undo` command is invoked, `ModelHistoryManager` returns a `ModelHistoryRecord`. A deep copy of the EntityLists contained within `ModelHistoryRecord` are then used to replace the EntityLists in the `ModelManager` for its operations, effectively reverting the data in Alfred to a previous state.

NOTE	The data in each `ModelHistoryRecord` in `ModelHistoryManager` is stored in memory, and is not stored on disc, so it will persist only while the Alfred application is running.

4.3.2. Implementation: How `ModelHistoryManager` Keeps Track of the State of the Data in Alfred

The following sequence diagram shows the sequence of method calls used to store the state of the data in Alfred in `ModelHistory` (`ModelHistoryManager` is an implementation of the `ModelHistory` interface) after the execution of a DeleteParticipantCommand:

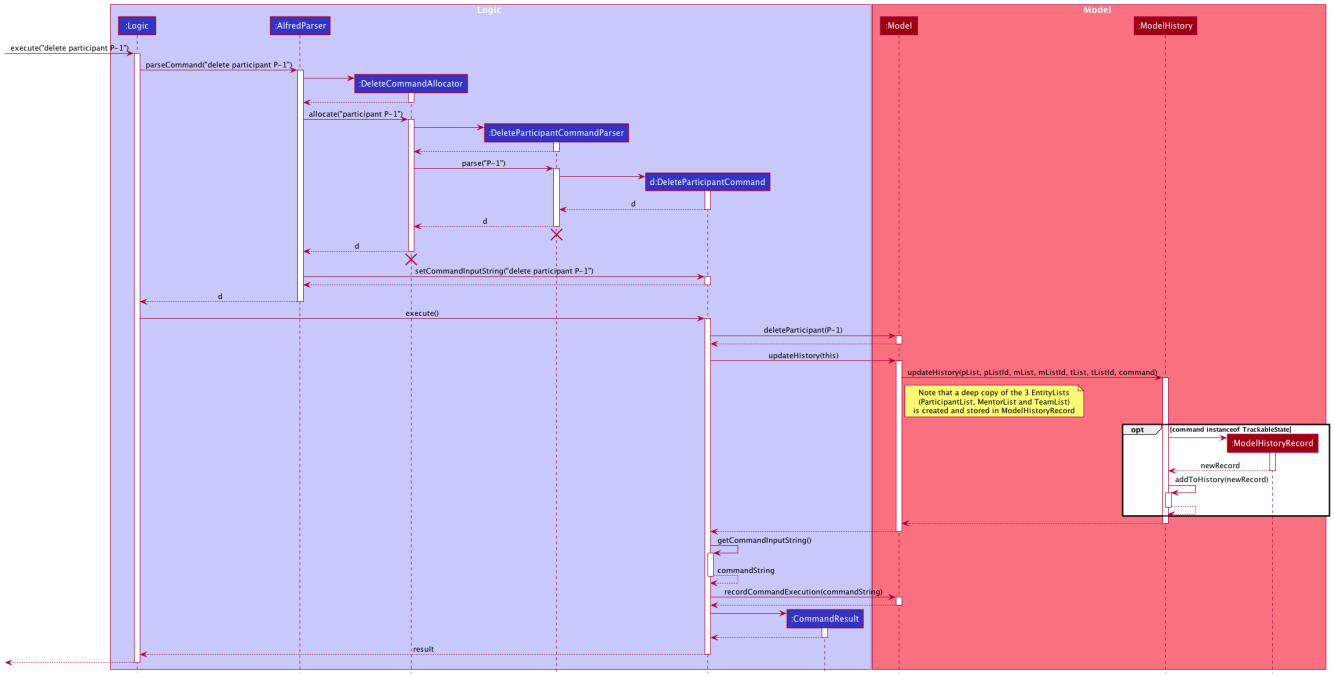


Figure 15. Sequence Diagram for the Updating of `ModelHistoryManager`

The top half of the diagram covers the creation of the DeleteParticipantCommand object, and the bottom half covers what happens when the `execute()` method of the DeleteParticipantCommand object is called. The important thing to note is the fact that a deep copy of the 3 EntityLists is created and stored as a `ModelHistoryRecord` in `ModelHistoryManager`.

An important issue to take note of is that only commands that implement the TrackableState interface will cause a new `ModelHistoryRecord` to be created and stored in `ModelHistoryManager`. The TrackableState interface is a marker interface, and is used to mark the commands that mutate data in Alfred. All command types except the following implement the TrackableState interface (and will therefore have the state of the data recorded in `ModelHistoryManager` after command execution): `help`, `list`, `find`, `history`, `leaderboard`, `getTop`, `export`, `help`, `home`, `undo`, `redo`.

4.3.3. Implementation: How `ModelManager` is Updated When the Undo Command is Executed

The following sequence diagram shows what happens when the UndoCommand is executed.

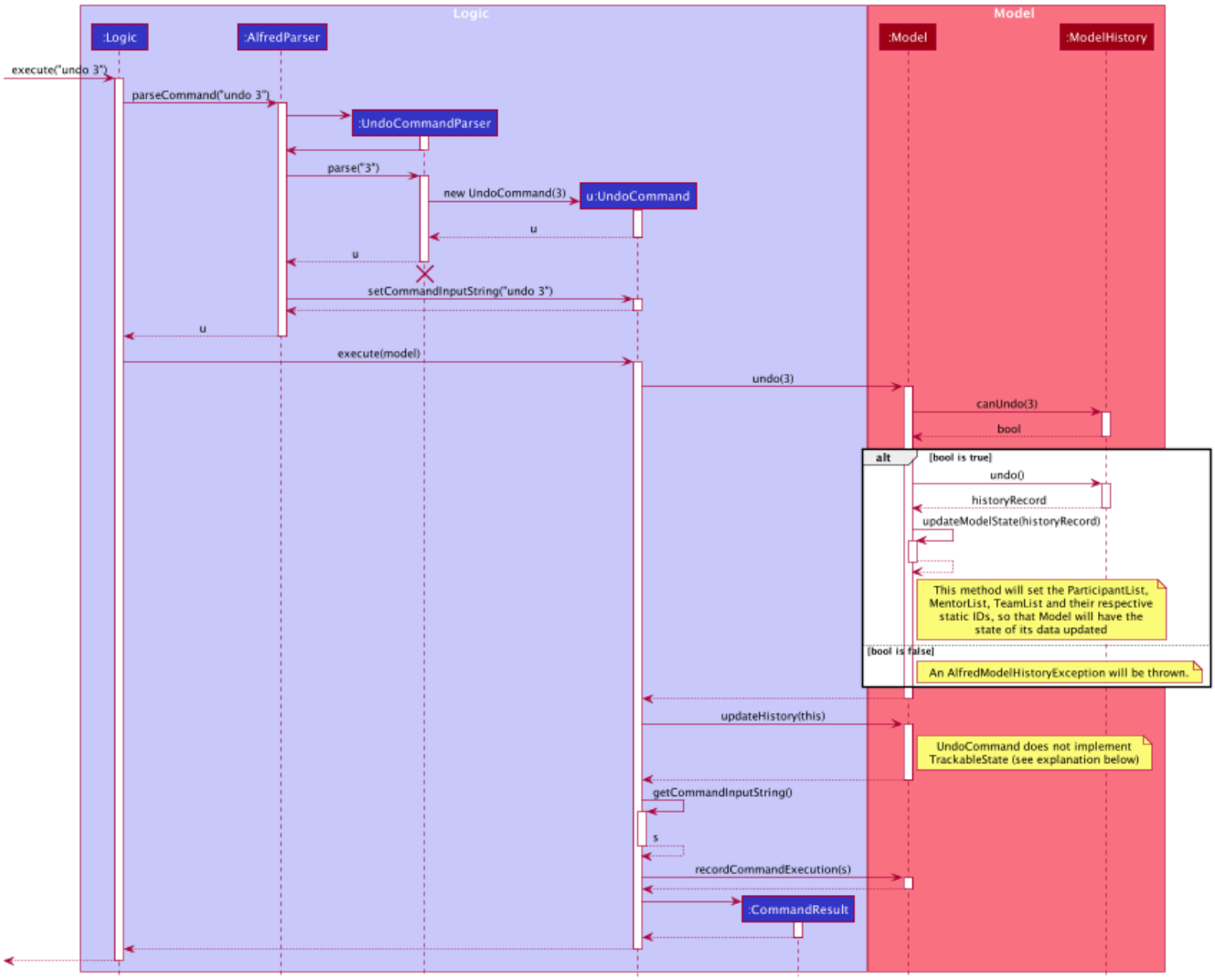


Figure 16. Sequence Diagram for the Execution of the Undo Command

The important issue to take note of here is that the code first checks whether it is valid to undo to a certain state by calling the `canUndo()` method in `ModelHistory`. The implementation of `ModelHistory` in `ModelHistoryManager` does so by checking if there are sufficient states to undo to, otherwise an exception is thrown.

NOTE	A analogous process is executed for the Redo Command.

4.3.4. Behaviour of Undo/Redo Mechanism

`ModelHistoryMangager` contains a List of `ModelHistoryRecord`, and a pointer pointing to the `ModelHistoryRecord` that reflects the current state of the data in Alfred.

In order to better illustrate how the state of the data is tracked and stored in `ModelHistoryManager`, consider the following example. The following commands are executed: 1. AddParticipantCommand: `add participant n/Clark Kent p/+6598321212 e/clark.kent@supermail.com` 2. AddMentorCommand: `add mentor n/Lex Luthor o/LexCorp p/+6598321010 e/lex.not.evil@gmail.com s/Social` 3. ListParticipantCommand: `list participants` 4. UndoCommand: `undo 2` 5. AddTeamCommand: `add team n/Justice League s/Social pn/BetterThanAvengers l/12`

This is the state of `ModelHistoryManager` when Alfred is first started.

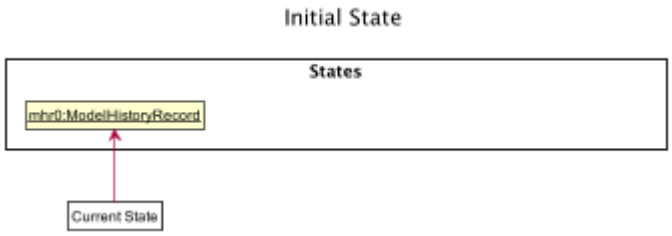


Figure 17. Initial State of `ModelHistoryManager`

This is what happens after each step:

Step 1. AddParticipantCommand: `add participant n/Clark Kent p/+6598321212 e/clark.kent@supermail.com`

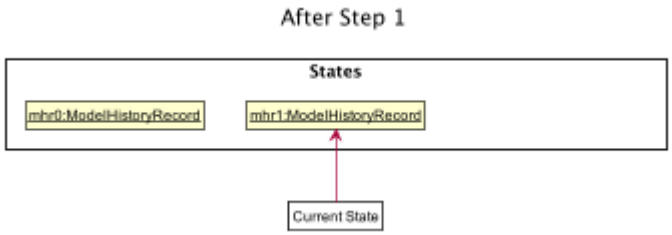


Figure 18. State of `ModelHistoryManager` *after Step 1*

A new `ModelHistoryRecord` is created to reflect the state of the data in Alfred after the execution of the AddParticipantCommand.

Step 2. AddMentorCommand: `add mentor n/Lex Luthor o/LexCorp p/+6598321010 e/lex.not.evil@gmail.com s/Social`

Figure 19. State of `ModelHistoryManager` *after Step 2*

A new `ModelHistoryRecord` is created to reflect the state of the data in Alfred after the execution of the AddMentorCommand.

Step 3. ListParticipantCommand: `list participants`

Figure 20. State of `ModelHistoryManager` *after Step 3*

Note that no new `ModelHistoryRecord` is created because the ListParticipantCommand does not alter the state of the data in Alfred. Hence, it does not implement the TrackableState interface.

Step 4. UndoCommand: `undo 2`

Figure 21. State of `ModelHistoryManager` *after Step 4*

After executing the `undo 2` command, the pointer in `ModelHistoryManager` shifts backwards by 2 to point to the `ModelHistoryRecord` at the zero-th index.

Note that this means that `undo 3` would throw an error, as you cannot move beyond the very first `ModelHistoryRecord` in `ModelHistoryManager`.

Step 5. AddTeamCommand: `add team n/Justice League s/Social pn/BetterThanAvengers l/12`

Figure 22. State of `ModelHistoryManager` *after Step 5*

Note that the execution of a new command will invalidate the `ModelHistoryRecord` after the pointer. This is because all subsequent data states are the result of transformations that have already been undone, so it is not valid to be able to `redo` to them.

4.3.5. Design Considerations

When designing the undo/redo feature, there were some design considerations to take note of.

Aspect: How Undo/Redo Executes

- **Alternative 1 (current choice):** Saves the entire data state of Alfred in memory.
  - 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, which is not trivial for certain commands, such as `import`, which provides a best-effort implementation and tries to import as many valid data entries as possible. In order to implement an `undo` method for this, we would have to keep track of the new Entities that got created due to the command execution and then invoke deletion of these Entities.

Given the large number of commands that are available in Alfred, it is not very scalable to

implement an undo/redo method for each of the commands. It is also more extensible to use Alternative 1 as it allows future commands to be added without the need for further changes for the undo/redo feature - simply get the new command's class to implement the TrackableState interface if it alters the state of the data in Alfred.

Aspect: Use of Marker Interface

Allows for an easy way to determine if the state of the data should be saved after the execution of the command. It is also very easy to change in the codebase. This means that should a feature in the future alter the state of the data in Alfred after execution, it is trivial to allow `ModelHistoryManager` to track the state.

Aspect: Limitation of Number of Data States Stored

Given that the Undo/Redo feature saves the state of the data in Alfred after the execution of TrackableState commands, it is important to ensure that memory usage by `ModelHistoryManager` is limited, otherwise Alfred will run very slowly and potentially crash once a substantial number of commands have been executed.

In order to accommodate this design for the Undo/Redo feature, we decided to limit the number of `ModelHistoryRecord` stored in `ModelHistoryManager` to 50. It is unlikely that a user would want to undo more than 50 commands at a go, as that would indicate a very significant error in the workflow, and recovering from that should not have a reliance on the Undo/Redo feature.

4.4. History feature

Closely related to the undo/redo feature, the `history` feature allows you to examine up to 50 previously executed commands in order to provide you with a visual understanding of the history of commands executed. Specifically, it provides you information on how many commands are undo-able/redo-able, and which commands are undo-able/redo-able. Otherwise, it can be difficult to know which commands you are undo/redo-ing, especially when many commands have been executed.

Simply execute `history` in Alfred to navigate to the "History" section of the Graphical User Interface and examine what commands are undo-able/redo-able.

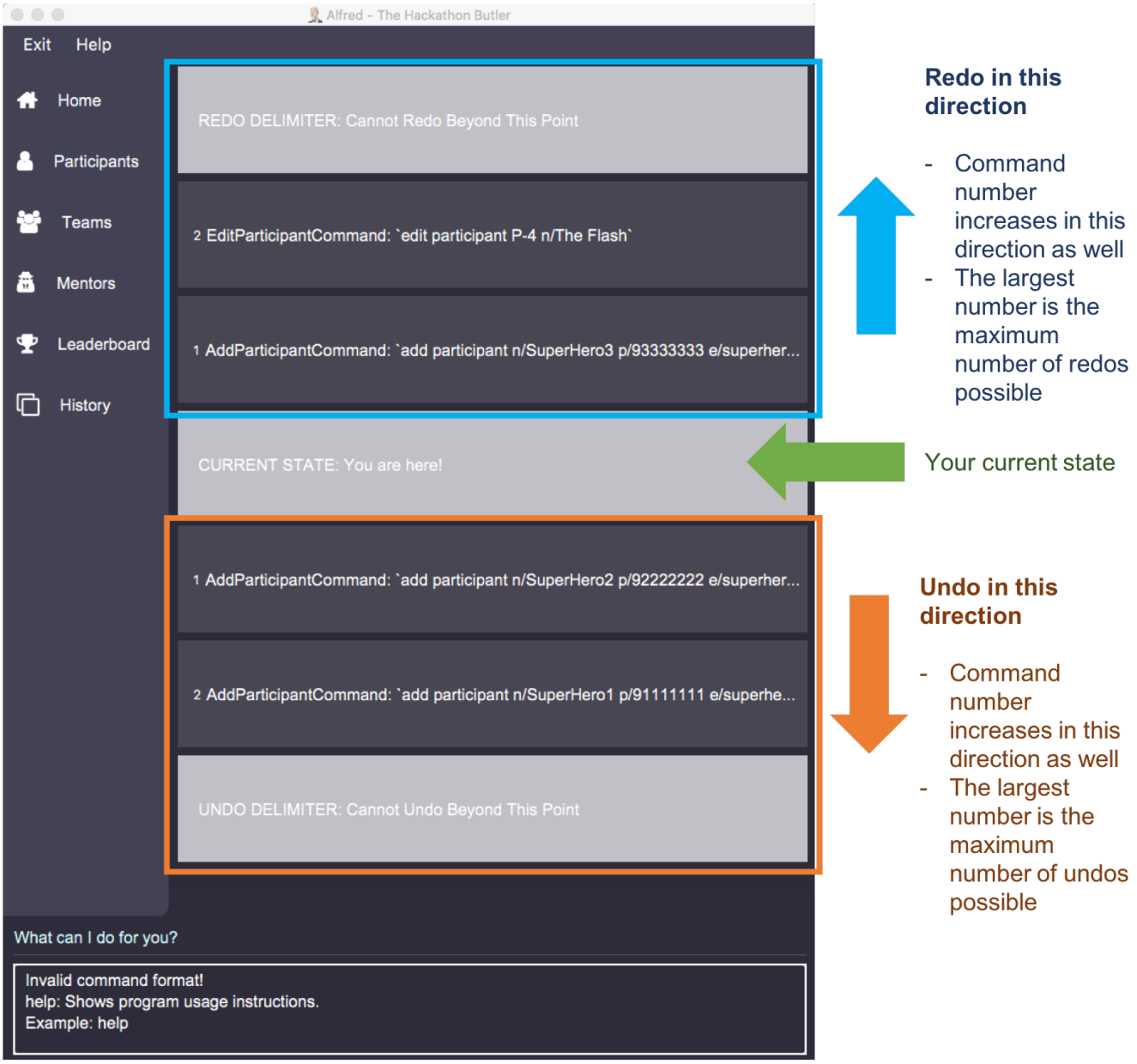


Figure 23. "History" section of the Graphical User Interface after invoking the `history` command

Note that there are 3 types of delimiters:

1. Redo Delimiter: No redos are possible beyond this point
2. Current Delimiter: This is the current state of the data relative to all the commands previously executed.
3. Undo Delimiter: No undos are possible beyond this point

4.4.1. Implementation

Most of the functionality required for the `history` feature is similar to that of the `undo`/`redo` feature. Specifically, there is a similar reliance on the `ModelHistory` interface and its implementation `ModelHistoryManager` to provide the information on which commands are undo-able and redo-able, along with their respective command input strings. See the sequence diagram below for more information.

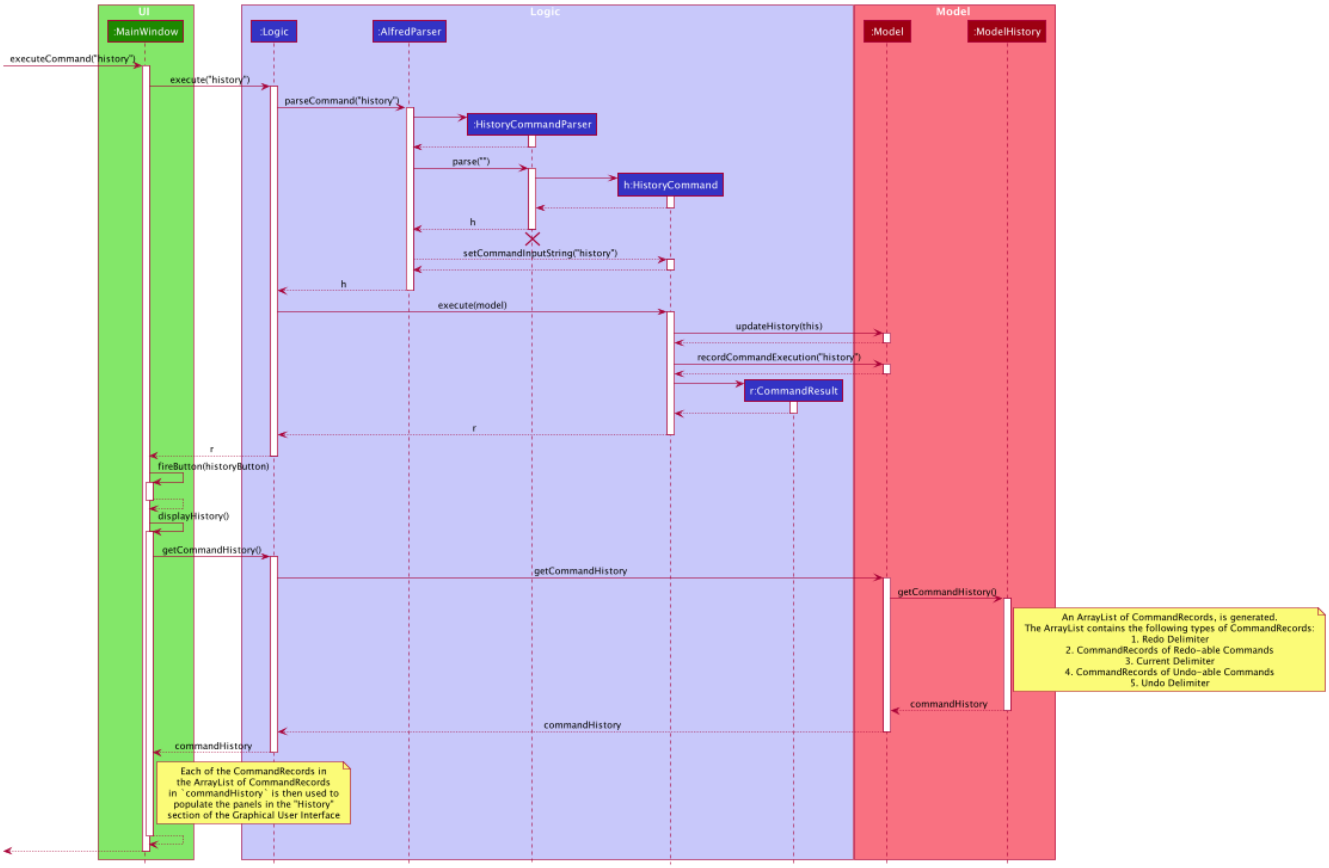


Figure 24. Sequence diagram for the execution of the History command

Since `ModelHistoryManager` keeps a linear history of Alfred's data state after commands are executed, and it has a pointer to the current data state, the redo-able commands are simply the ones after the pointer, and the undo-able commands are the one before the pointer. The panels displayed in the "History" section of the Graphical User Interface is simply a visual representation of this sequential ordering of datas states. The call to `getCommandHistory()` in Model will return an ArrayList of CommandRecords, where each CommandRecord specifies the command input string as well as whether the CommandRecord is a delimiter. This ArrayList is then provided back to the UI for rendering in the "History" section of the Graphical User Interface.

4.4.2. Design Considerations

Given the close relationship between this feature and the `undo`/`redo` feature, the design considerations are very similar as well. See Section 4.3, "Undo/Redo feature" for more information.

4.5. Command History Navigation

Since Alfred is ultimately a Command-Line Interface (CLI) application, it would be good to integrate some of the more useful features of CLI applications into Alfred. This feature allows users to go through previously executed commands by pressing the ALT+UP/DOWN arrow keys. Although it would be more convenient to use the UP/DOWN arrow keys to navigate the command history, the UP/DOWN arrow keys have been mapped to the Command Suggestions feature in Alfred.

This feature allows users to quickly re-use previously exeuted commands without having to go through the hassle of re-typing everything. This is particularly useful when the commands are long, and only small modifications are necessary to the command.

NOTE

Only successfully executed commands can be navigated to using the ALT+UP/DOWN keys. Invalid comands will not be stored, and hence cannot be accessed using this feature. The only exception to this is the `import` command, as it is a 'best-effort' command that raises exceptions for certain entries in the csv file, but seeks to import as many valid data entries as possible. Hence, the `import` command can be navigated to using the ALT+UP/DOWN arrow keys even when it does not completely succeed during execution.

4.5.1. Implementation

Every time a valid new command is executed, the string used to generate the command (i.e. the text that the user types into Alfred's Command Input Box) is stored in the `Command` object.

The main class responsible for remembering and providing the previously used command input strings is the `CommandHistoryManager` class, which implements the `CommandHistory` interface. The `CommandHistory` interface only exposes 3 methods: `saveCommandExecutionString`, `getPrevCommandString` and `getNextCommandString`. The latter 2 methods directly map to the 2 situations of pressing the ALT+UP and pressing the ALT+DOWN keys respectively.

When the `Command` object is executed, that string is then stored in the `CommandHistoryManager`. A linear list of successfully executed commands' input strings is stored in `CommandHistoryManager`, and a pointer to the current string being displayed in the textbox is used to indicate which is the currently active string.

NOTE

Only the last 50 commands are stored in `CommandHistoryManager`. Anything beyond that is discarded.

4.5.2. Implementation: Setting and Storing of Command Input String

The following sequence diagram describes the sequence of method calls used to set and store the command input string in `CommandHistoryManager`

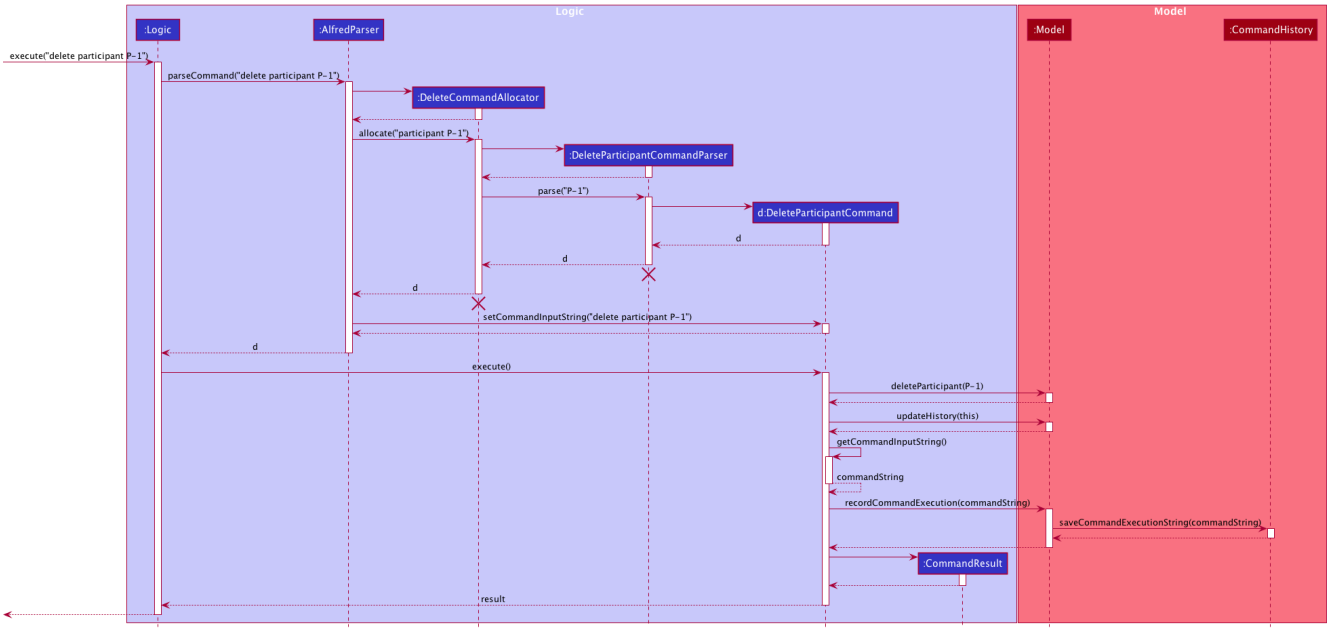


Figure 25. Initial State of `ModelHistoryManager`

4.5.3. Implementation: Arrow Key Invocation

The following sequence diagram describes the sequence of method calls used to set the text in Alfred's Command Input Box whenever the ALT + UP/DOWN arrow keys are pressed.

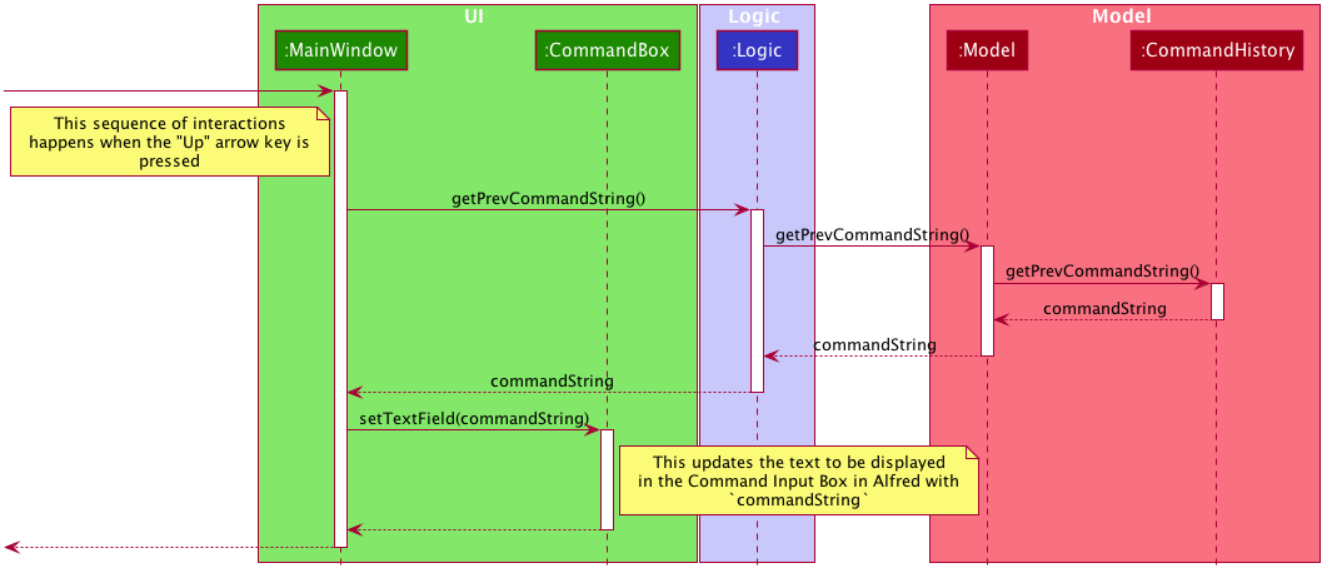


Figure 26. Sequence Diagram for when ALT+UP arrow keys are pressed

An analogous process is executed when the ALT+DOWN arrow keys are pressed.

4.5.4. Design Considerations

The following are some design considerations for the Command History Navigation feature.

Aspect: Use of CommandHistoryManager

A separate class was created for managing the command history. Although this meant that Model would have a further dependency on another class and hence cause an increase in the coupling between objects, this implementation adheres to the Single Responsibility Principle and abstracts away the details of handling the command history to a purpose-built class. This adheres better to the OOP-style of programming and results in conceptually cleaner code.

Aspect: Limitation of Command History Size

In order to ensure that `CommandHistoryManager` does not occupy an increasingly large portion of memory as more and more commands are executed, it is necessary to limit the number of commands that are stored. Otherwise, Alfred's performance will suffer as more commands are executed.

Furthermore, it is very unlikely for user to want to navigate more than 50 commands into history, as it would likely be more convenient to re-type the command if that is the case.

4.5.5. Aspect: Usage of Arrow Keys

Most CLI applications with such a Command History Navigation feature would use the UP/DOWN arrow keys directly, but the UP/DOWN arrow keys have been mapped for a different use (Command

Suggestion Feature) in Alfred. Hence, we decided to use the ALT modifier key for the feature. It is an inconvenience, but likely a minor one.

4.5.6. Aspect: Navigation to Successfully-Executed Commands

As mentioned above, only successfully executed commands can be navigated to using this feature. This is because unsuccessful commands will still remain within the Command Input Box in Alfred, which the user can readily edit. Only upon successful command execution will the text in the Command input Box disappear, so this feature is necessary to retrieve the Command's input text.

4.6. `home` feature

Entering the `home` command will prompt the `Logic`to generate a `Statistics` object. The `Statistics` object is generated by obtaining entity lists of different types(ParticipantList, TeamList, MentorList) from `Model`. The respective lists are then converted to the Stream data structure and filtered through to obtain the distribution of each entity type by `Subject`, as well as the total number of each entity. These information is stored in the `Statistics` object. The `Statistics` object is then used to construct a `StatisticsListPanel`, where the information is parse to be displayed as a distribution bar graph.

The following sequence diagram shows how the `home` operation works:

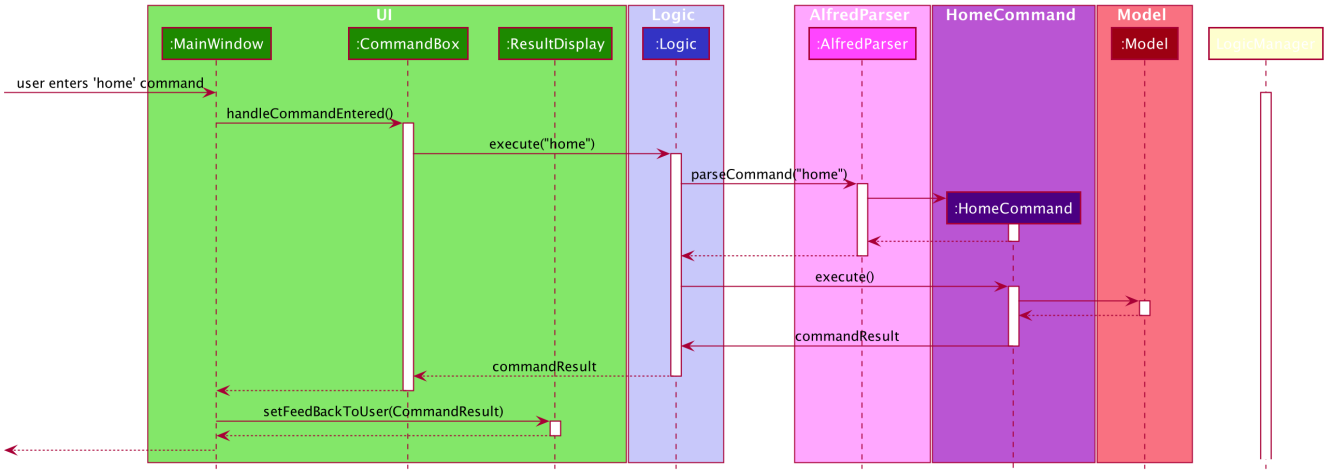


Figure 27. Sequence Diagram for `home` command

4.6.1. Design Considerations

1. Finding the distribution of Teams and Mentors by subjects
    - Alternative 1: Implementing methods to keep track of number of mentors or participant under the respective TeamList or ParticipantList class.
        - Pros: Each of TeamList or Participant list will have their individual responsibilities in keeping track of the distribution of its teams or mentors.
        - Cons: Clutters up the Model and ModelManger interface.
        - Cons: Does not make use of existing methods under Model.
    - Alternative 2: Getting TeamList or ParticipantList from Model and implementing operation to find the distribution separately.

- Pros: Makes use of existing Model methods of getting TeamList and ParticipantList.
- Pros: Does not clutter up Model and ModelManager with different methods
- Pros: Greater flexibility in implementing methods to find the distribution of teams or mentors.
- Cons: Clutters up the Model and ModelManger interface.
- Cons: Does not make use of existing methods under Model.

Decision: We decided to proceed with this alternative 2 because it makes use of existing methods under Model and provides greater flexibility on how I can find the distribution number by subject from the TeamList or ParticipantList.

4.7. Command suggestion feature

This feature provides suggestions by predicting the commands that a user intends to enter(as the user types).

NOTE	Only suggestions that start with the same alphabets or spaces as those entered by user will be suggested.

4.7.1. Implementation

The main class responsible for remembering and providing the previously used command input strings is the `AutoCompleteCommandBox` class.Typing into the `AutoCompleteCommandBox` will prompt the attached `Listener` to be activated. Activation of the `Listener` will prompt it to filter through the set of predefined command suggestions. The commands that start with the same text entered by user will be filtered through. The first five result will then be mapped to their respective `TextFlow` object and added to the `ContextMenu`. This `ContextMenu` will then appear as a pop up box. ser input box, the method. Pressing `up` and `down` arrow keys will enable the user to choose a command suggestion. Additionally, pressing `enter will filter out different `Text` from the `TextFlow` object. The `setText` method will then be called to set the `JFXTextField` to the said `Text` object.

4.8. `assign` / `remove` feature

Upon successful assignation, the new participant or mentor will be stored internally in the list of participant or optional mentor field in the Team object. Upon successful removal of Participant or Team, the specified participant or mentor will be removed from the Team object. Additionally, it calls the following operations:

- `ModelManager#addMentorToTeam` – adds mentor to a specified team
- `ModelManager#addParticipantToTeam` – adds participant to team
- `ModelManager#removeParticipantFromTeam` – removes participant from team
- `ModelManager#removeMentorFromTeam` – removes mentor from team

1. The `assign participant` command will add the new participant under the list of participant in the specified `Team` object. This is provided that the number of members in the `Team`

object(size of list of participants) is less than 5.

The following sequence diagram shows how the `assign participant` operation works:

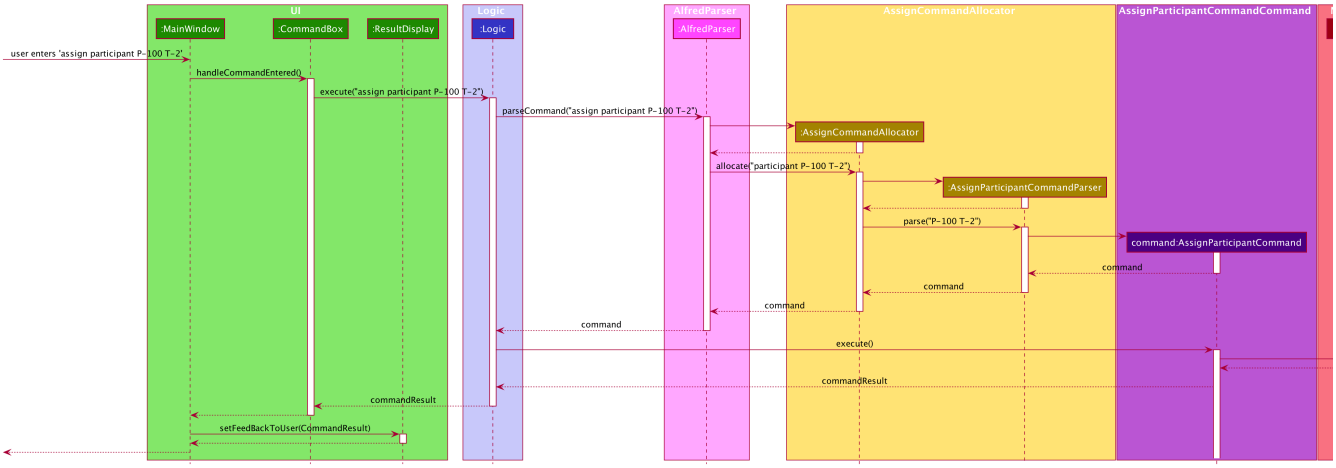


Figure 28. Sequence Diagram for an example of `assign participant` command

1. The `assign mentor` command will add the new mentor under the `Optional<Mentor>` field in the specified `Team` object. This is provided that there is no existing mentor in the team. The sequence diagram of `assign mentor` is similar to that of `assign participant`.
2. The remove participant will first search through the list of participant under the specified `Team` object. This checks whether the specified participant is a member of the team in the first place. If it is not a member, an error will be thrown. Whereas if it is a member, the specified participant will be removed from the list of participant.

The following sequence diagram shows how the `remove participant` operation works:

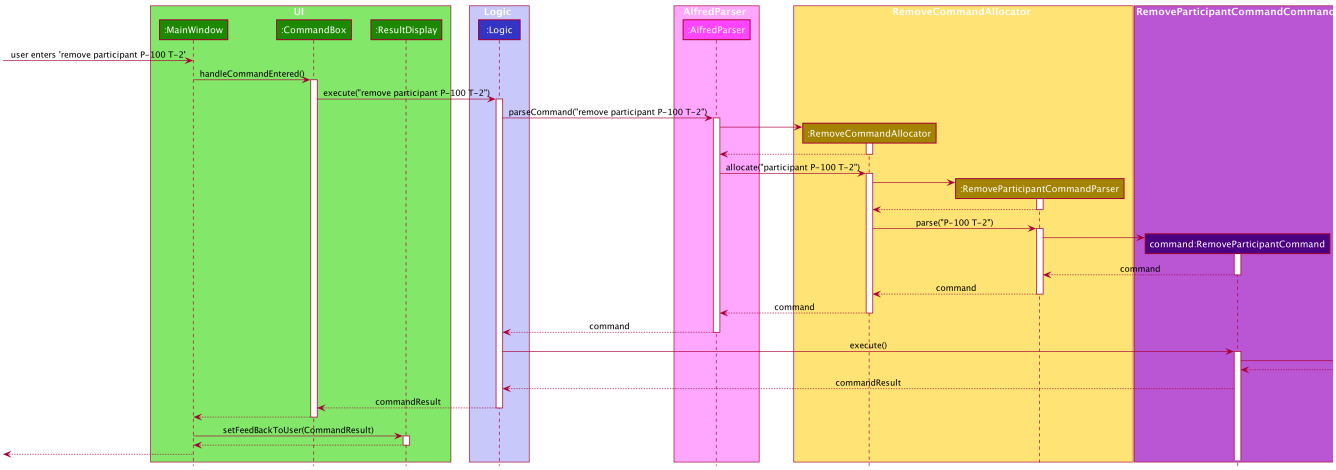


Figure 29. Sequence Diagram for an example of `remove participant` command

1. The `remove mentor` command will first check whether the `Optional<Mentor>` field under the specified `Team` object is not empty and corresponds to the specified mentor. This checks whether the team have not been assigned a mentor, or they have been assigned to a different mentor. Under any of these cases, and error will be thrown. Whereas if the team is assigned the specific mentor, the specified mentor will be removed from the `Optional<Mentor>` field.The sequence diagram of `remove mentor` is similar to that of `remove participant`.

4.9. Scoring

As its name suggests, this feature's intended purpose is to aid users in the process of giving scores to the teams participating the hackathon. The `score` command is a crucial feature of Alfred as judging and scoring are core activities within every hackathon. Alfred provides the following implementations of the `score` command:

1. `score add`: allows users to add a certain number of points to a team's score
2. `score sub`: allows users to subtract a certain number of points from a team's score
3. `score set`: allows users to set a team's score to a certain number of points
4. `score reset`: allows users reset a team's score to 0

4.9.1. Implementation Overview

This feature and its varieties have been implemented in a relatively straightforward manner, as the Class Diagram below showing the high level representation of the Object Oriented solution devised to implement this feature highlights.

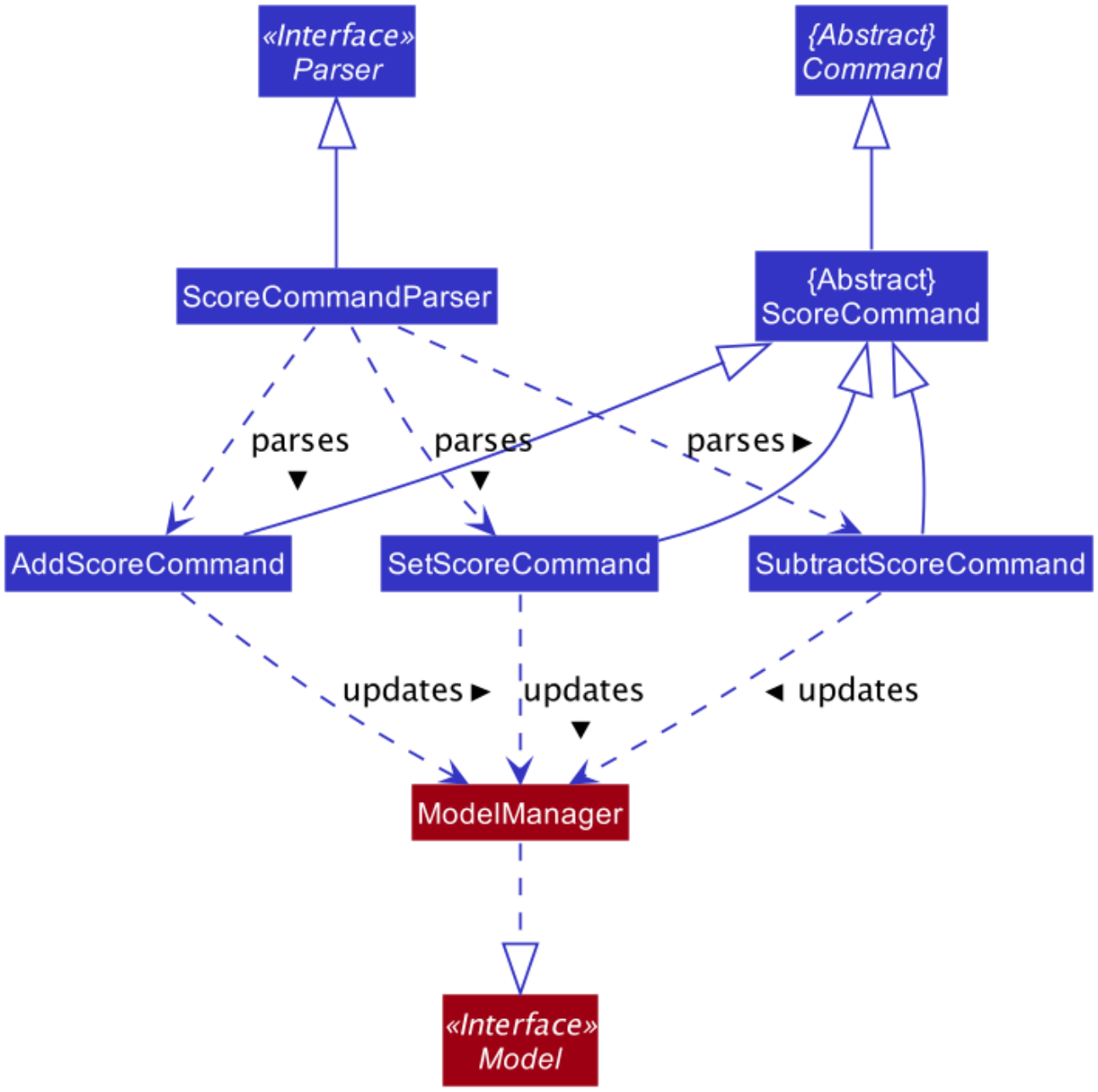


Figure 30. Scoring Feature Class Diagram

From the above diagram it can be seen that each different implementation of the `score` command inherits from the same `ScoreCommand` abstract class. The `ScoreCommand` abstract class provides a base for the implementation of the current specific `score` commands and in the future any further additions made to the `score` command functionality must also follow this same convention.

Secondly, there is no `ResetScoreCommand` class. This is done intentionally as the `SetScoreCommand` can be reused to reset a particular team's score, thereby making better use of abstraction.

A representation of how the above classes interact to provide execute a user's command is highlighted in the sequence diagram below. This sequence diagram illustrates the object interactions when a user types the command `score add T-1 40`. For context, this command adds 40 points to the team with ID "T-1".

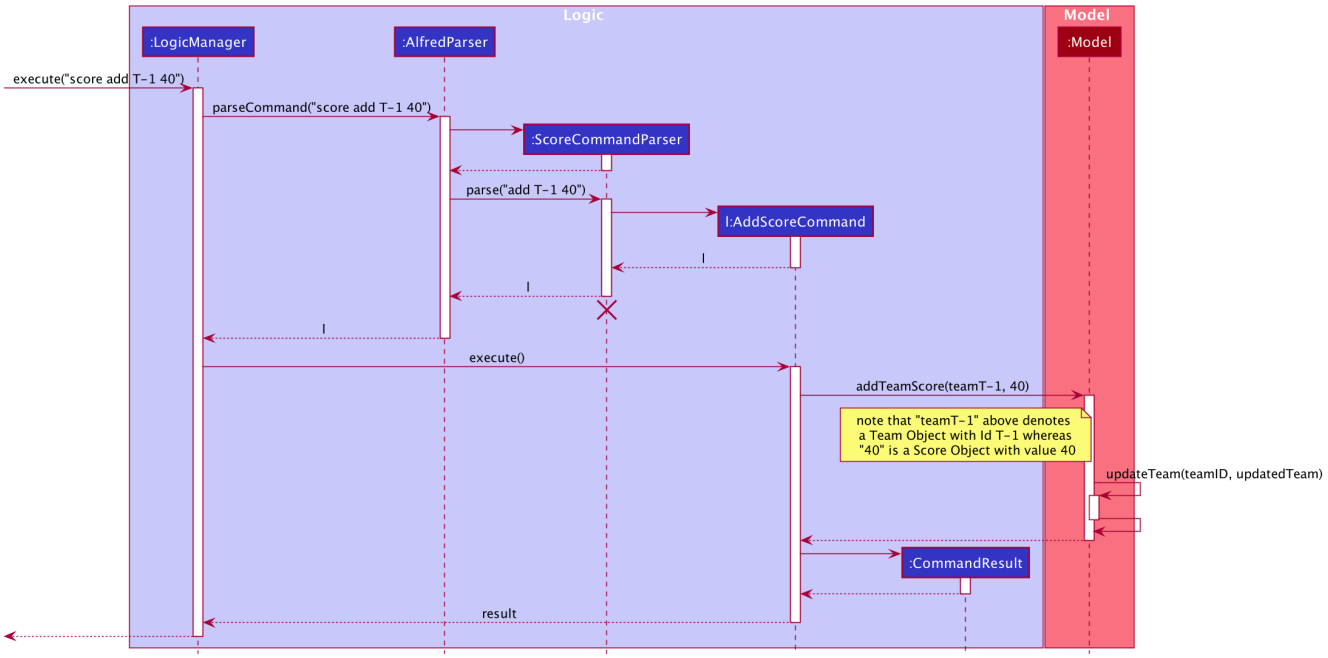


Figure 31. Add Score Command Sequence Diagram

As seen from the above diagram, the execution of the `score add` command can be put in simple words as per the following steps:

- Step 1: `LogicManager` starts executing the user's command and calls the `AlfredParser` to parse it.
- Step 2: `AlfredParser` find the appropriate `Parser` to parse the command and creates a new `ScoreCommandParser` to parse the arguments of the score command, essentially "add T-1 40".
- Step 3: The `ScoreCommandParser` then parses the arguments and is responsible for:
  - Checking which specific implementation of the `score` command is being called (in this case "add") and ensuring it is a valid method.
  - Parsing the Team ID specified by the user and ensuring it is of a correct format.
  - Parsing the Score specified by the user and ensuring it is a valid score.
- Step 4: `ScoreCommandParser` then creates a new `AddScoreCommand` object with the above parsed Team ID and Score, which is then returned all the way to `LogicManager`
- Step 5: `LogicManager` then executes the `AddScoreCommand` object upon which the `AddScoreCommand` object calls `Model` 's `addTeamScore()` method which updates the specific team's score within `Model`. The updating is done by the `updateTeam(teamID, updatedTeam)` method of `Model` which updates the Team object with ID "teamID" within `Model` to match the "updatedTeam" team object.
- Step 6: Upon updating the team's score in `Model`, the `AddScoreCommand` object creates a new `CommandResult` object which is returned to the UI component (not shown in the diagram) to display a feedback message to the user.

Though the above diagram and steps are designed in the context of a `score add` command, the logic applies to every other type of `score` command as well. The only difference is that the `ScoreCommandParser` creates the appropriate command object for the command and each command object calls a different method from `Model`, as per the following:

1. `score subtract`: `ScoreCommandParser` creates a new `SubtractScoreCommand` instead of an `AddScoreCommand` and this `SubtractScoreCommand` object calls the `subtractTeamScore()` method of

`Model`.

2. `score set` and `score reset`: `ScoreCommandParser` creates a new `SetScoreCommand` instead of an `AddScoreCommand` and this `SetScoreCommand` object calls the `setTeamScore()` of method `Model`.

4.9.2. Design Considerations

To develop the `score` feature a few considerations and decisions had to be made with regards to how to implement the feature at various steps. This section focuses on some of the aspects wherein we faced dilemmas and how we addressed them.

Aspect: How to implement the Score Commands

Currently, Alfred's score commands all inherit from a single `ScoreCommand` abstract class rather than each command class inheriting directly from the command class. This was done in order to provide a concrete base for the score commands, but more importantly also to make better use of polymorphism within our codebase. This use of polymorphism allows the Alfred codebase to avoid several code repetitions, for example by facilitating the use of a single parser class to parse the `score` commands as it can then simply return a `ScoreCommand` object regardless of which specific `score` command is called. Thanks to this, we avoid having to make a parser for every single `score` command.

Aspect: How to implement `score sub` (Subtract Score Command)

While traditionally a subtract command would be implemented using an add command only with negated values, this is not quite possible in Alfred's case. Within Alfred `Score` objects have a strict restriction that they cannot be created with negative values as a team's score can never be negative. This restriction cannot be relaxed as it ensures that any data being imported into Alfred does not violate this property either. Due to this the `score sub` command could be implemented by reusing the `AddScoreCommand` and hence why it has its own command class `SubtractScoreCommand`.

Aspect: Implementation of `score reset` command

- **Alternative 1:** Create a `ResetScoreCommand` class
    - Pros: Lowers confusion as all the relevant code is in its own class rather than mixed with other code.
    - Cons: Leads to duplication of code as the command is very similar to the `score set` command, making it poor software engineering practice.
- **Alternative 2 (Current Choice):** Reuse the `SetScoreCommand` class
    - Pros: Better use of abstraction and reduces the amount of duplicate code written.
    - Cons: Overcrowds a single class as feedback messages and other properties of the `score reset` command also need to be written within it.

Upon close inspection it was observed that the removal of duplicate code would far outweigh the convenience and orderliness of having a separate class for the command, especially considering that a `ResetScoreCommand` would have far more duplicate code than unique code as compared to the `SetScoreCommand`. Bearing this in mind, we ultimately decided to choose "Alternative 2" as it would allow for better reuse of existing code and follow better software engineering practices.

4.10. Leaderboard and Get Top Teams

The `leaderboard` and `getTop K` commands are two very important features of Alfred as they allow the user to automatically sort the teams by their scores, fetch any number of top teams in the competition and identify and break ties between teams conveniently. The execution of either of these commands displays the resultant teams on the UI in their correct sorted order. The following subsections explore the implementation of each of these commands and provide an insight into the design consideration made when developing them.

4.10.1. Implementation Overview

The implementation of these two commands is very similar in nature. They both:

- rely on updating a `SortedList` of teams present within the `ModelManager` class, which will be referred to as `sortedTeamList` in subsequent sections. This list is used to display the command's results on the UI.
- use an ArrayList of `Comparator<Team>` objects to contain additional comparators. These are used to break ties between teams on a basis other than score.

The class diagram below provides a high level representation of the Object-Oriented solution devised to implement the aforementioned features.

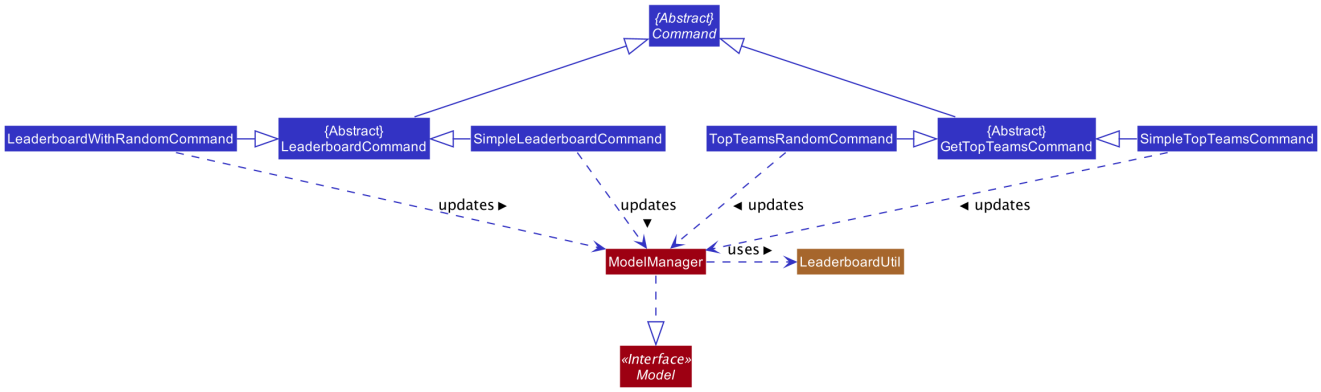


Figure 32. Leaderboard and Get Top Teams Implementation Overview

From the above class diagram, there are two important matters to note regarding the implementation of these features:

1. The `LeaderboardCommand` and `GetTopTeamsCommand` are implemented as abstract classes which extend the `Command` abstract class. Any command to do with leaderboards or getting the top teams extends either one of these abstract classes depending on which command it is.

2. The `ModelManager` class uses another class `LeaderboardUtil` which provides utility methods for the Leaderboard and Get Top Teams commands, such as fetching an appropriate number of teams for the `getTop K` command and breaking ties between teams for both commands.

With the class structure covered, the following sub-sections seek to explain how the different classes in Alfred interact to produce a result for the user, and finally the design considerations that were made for each command.

4.10.2. Leaderboard Command Implementation

The `leaderboard` command fetches a leaderboard consisting of all the teams registered for the hackathon, in descending order of their score. Additionally, if tiebreak methods are specified, ties between the teams are typically broken in one of two ways:

- **Comparison-based tiebreakers:** wherein the user picks certain tiebreak methods which rely on comparing certain properties of teams, such as the number of participants they have.
- **Non-Comparison-based tiebreakers:** wherein the user breaks ties on non-comparison based methods (currently only the "random" method) in addition to any Comparison-based tiebreakers.

Given below is the sequence diagram illustrating the flow of events which generates a result for the user when he types the command `leaderboard tb/moreParticipants`. For your reference, here the prefix "tb/" is used to precede a tie-break method and "moreParticipants" is a tie-break method which gives a higher position to teams with more participants. Essentially this demonstrates the flow for a "Comparison-based tiebreak".

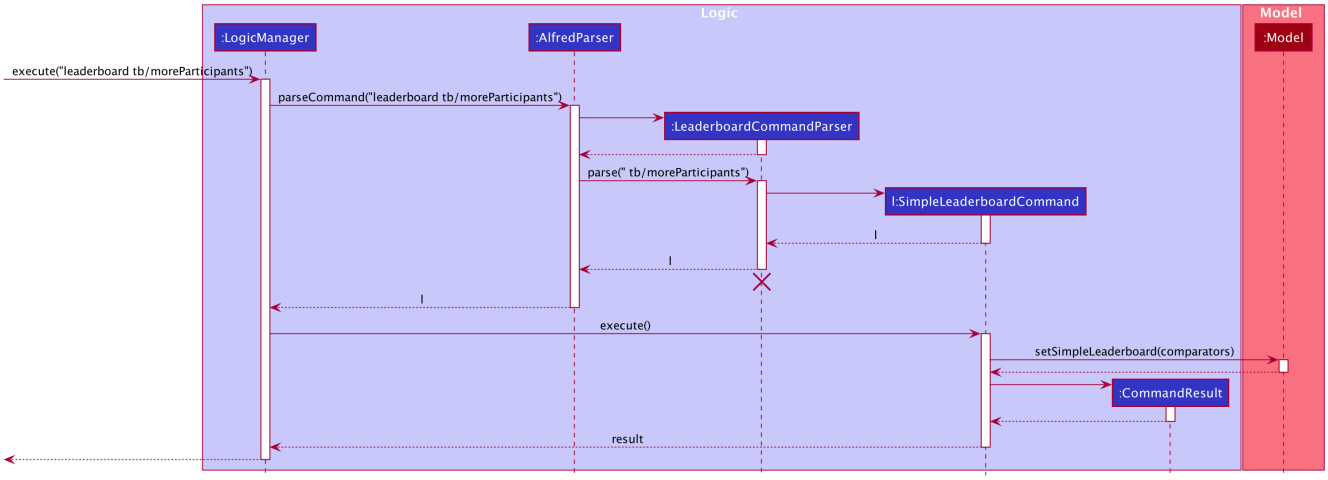


Figure 33. Interactions within Logic Component for SimpleLeaderboardCommand

In the above, the `LeaderboardCommandParser` class parses the tie-break part of the command, particularly "tb/moreParticipants". Based on this input, it creates a new `ArrayList<Comparator<Team>>` object and appends the appropriate comparators to it based on the specified tiebreak methods. A new `SimpleLeaderboardCommand` object is then created with this array list as its parameter and returned.

When the `SimpleLeaderboardCommand` is then executed, it calls `Model` 's `setSimpleLeaderboard(comparators)` method with the input parameter being the `ArrayList<Comparator<Team>>` passed as parameter for the former's creation.

`Model` 's `setSimpleLeaderboard(comparators)` method updates the `SortedList` of teams within `Model` itself, which is then displayed on the UI when the new `CommandResult` object is created and returned.

This flow of events, albeit a few differences, is the same for every variation of the `leaderboard` and `getTop K` commands explored subsequently.

Do note that if the user's input did not specify any tie-break methods, hence just being `leaderboard` then the `SimpleLeaderboardCommand` object would be created with an empty ArrayList of comparators.

The flow of events for this particular scenario would be unchanged from the above illustration.

However, it often occurs that even tiebreak methods cannot separate two teams in a hackathon, for which organizers randomly select a winner from the tied teams, basing it purely on fair luck. The `leaderboard` command with the tiebreak method `random` is used to provide this functionality.

Given below is the sequence diagram illustrating the flow of events which generates a result for the user when he types the command "leaderboard tb/moreParticipants random". For your reference, here the prefix "tb/" is used to denote a tie-break method and "moreParticipants" is a tie-break method which gives a higher position to teams with more participants, and "random" is another non-comparison based tie-break method.

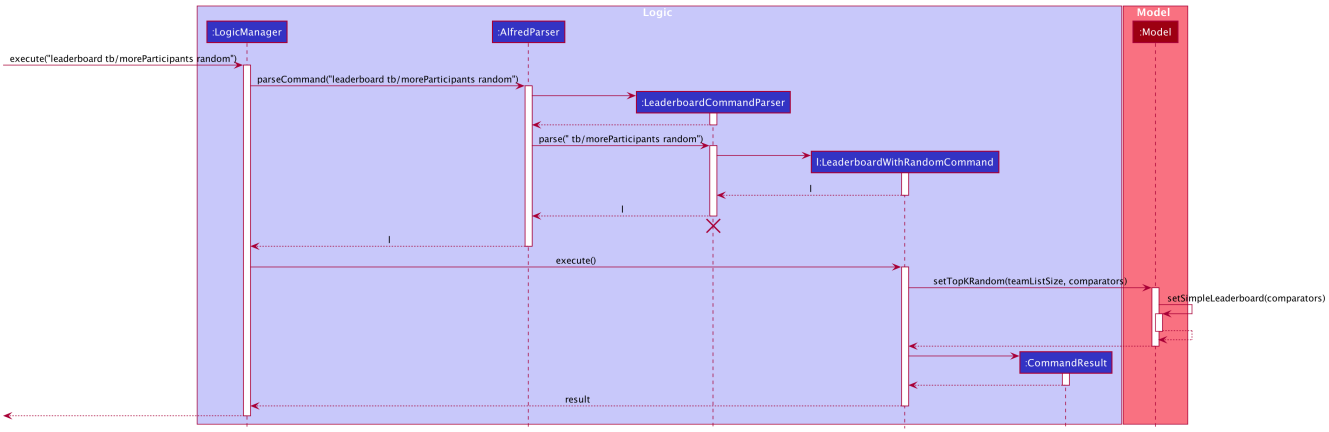


Figure 34. Interactions within Logic Component for LeaderboardCommand with Random Winners

The above sequence follows the exact same logic as that for the Simple Leaderboard as explained above (See [SimpleLeaderboard-Explanation]).

However, in this case the `LeaderWithRandomClass` calls the `setTopK(teamListSize, comparators)` method of `Model` which essentially breaks any remaining ties after applying the tie-break methods between teams on a random basis, and fetches a number of teams equal to `teamListSize` which is the size of the `sortedTeamList` reflecting the total number of teams in the hackathon.

Secondly, `Model` calls its own method `setSimpleLeaderboard(comparators)`, which essentially resets the `sortedTeamList` of teams clearing it of any sorting, and then applies the new comparators to it, before the algorithm for random winners can be applied to the `sortedTeamList`.

4.10.3. Leaderboard Design Considerations

There were several questions we asked ourselves over the course of developing the leaderboard feature. The following contains certain aspects we had to consider during the development stage and details how and why we decided to use a certain methodologies over others.

Aspect: How to store the sorted list of participants

- **Alternative 1:** Use the existing List in `ModelManager` storing the teams.
  - Pros: Easier to implement as lesser extra code involved, as most getters and setters have already been coded.
  - Cons: Sorting will be more complicated and potentially slower with large number of teams

as the other lists are `FilteredList` objects, whose API doesn't allow direct sorting.

    - Cons: An existing List is likely to be used by other commands to display data on the UI, so with any sorting will have to undone each time after use; a process which is prone to careless errors.

- **Alternative 2 (Current Choice):** Use a new `SortedList` object from the JavaFX Library
    - Pros: Easy and quick to sort contents with the `SortedList` API.
    - Pros: A new list means the sorting will not interfere with any other feature's operations, such as the `list` command which uses the existing `filteredTeamList` holding all the teams.
    - Cons: Another List to handle in `ModelManager` which increases the amount of code.

Due to the overwhelming benefits and conveniences that a new `SortedList` of teams would bring in the development of Alfred's `leaderboard` and `getTop K` commands, we decided to rely on "Alternative 2" with regards to this dilemma.

Aspect: Designing Leaderboard's Command Classes

- **Alternative 1:** Use a single `LeaderboardCommand` class
    - Pros: Lesser duplicate code as both ("random" and "non-random") tiebreak methods can be handled within a single class.
    - Cons: Introduces control coupling as the `LeaderboardCommandParser` will have to send a flag to `LeaderboardCommand` to indicate whether "random" should be applied or not as a means of tie-break.
- **Alternative 2 (Current Choice):** Use an Abstract `LeaderboardCommand` class inheriting from `Command` which any `leaderboard` related commands will themselves extend.
    - Pros: Single Responsibility Principle will be better respected as any change in logic for one type of `leaderboard` command will only affect its respective class. Secondly, no longer a need for a flag as the parser can directly call the appropriate command class.
    - Cons: Introduces slight duplication in code as each class will contain a similar segments of code for checking the status of the teams in `Model`.

We decided to follow "Alternative 2". Firstly, if a single class were being used, it would be difficult to distinguish which type of `leaderboard` command should be called - whether a leaderboard with or without "random" as tiebreak should be used. This would require the `LeaderboardCommandParser` to pass a flag signalling whether the "random" version should be called or not, which introduces control coupling. Although with a single distinct method (ie "random") this seems manageable, as the scale of Alfred increases with more non-comparison based methods such as "random" being introduced, passing a flag from `LeaderboardCommandParser` to the `Leaderboard` command class would become less and less manageable. Secondly, we wanted to avoid coupling the `Parser` and `Command` classes in a way which `Parser` influences the behaviour of the `Command` as it introduces leeway for errors.

Aspect: Where to Write Algorithms used by `leaderboard` (and `getTop NUMBER`) Command

- **Alternative 1:** Write the methods as private within `ModelManager` itself
    - Pros: Relevant code is in close proximity to where it is being called allowing for easy

reference of what is being done and quick rectification if needed.

    - Cons: Would harm Single Responsibility Principle as `ModelManager` would need to be changed in case there is change in required to the Leaderboard Algorithms, whereas it should only be changed if there is a change required to `Model`

- **Alternative 2 (Current Choice):** Create a new `LeaderboardUtil` class
    - Pros: Maintains single responsibility principle and ensures greater abstraction as complicated algorithms are simply handled by another class altogether.
    - Cons: Increases the amount of coding and documentation required. Additionally, it brings about the inconvenience of having to shift between classes to view the available methods and their implementations.

"Alternative 2" was eventually selected as it follows better Object-Oriented Programming practices. By abstracting away the methods used to sort and tie-break teams and keeping them in another class, the overall readability of the code is enhanced and would be easier for any future programmers working on this project to understand and work on.

4.10.4. Get Top Teams Implementation

The `getTop K` command fetches the top "K" number teams sorted in descending order of their points, where K is a positive integer inputted by the user. The `getTop K` command follows a similar pattern as the `leaderboard` command in the sense that ties between teams are broken in one of two ways:

- **Comparison-based tiebreakers:** wherein the user picks certain tiebreak methods which rely on comparing certain properties of teams, such as the number of participants they have.
- **Non-Comparison-based tiebreakers:** wherein the user breaks ties on non-comparison based methods (currently only the "random" method) in addition to any Comparison-based tiebreakers.

Given below is the sequence diagram illustrating the flow of events which generates a result for the user when he types the command "getTop 3 tb/moreParticipants". For your reference, here the prefix "tb/" is used to denote a tie-break method and "moreParticipants" is a tie-break method which gives a higher position to teams with more participants. This essentially reflects a tie being broken by comparison-based tiebreakers.

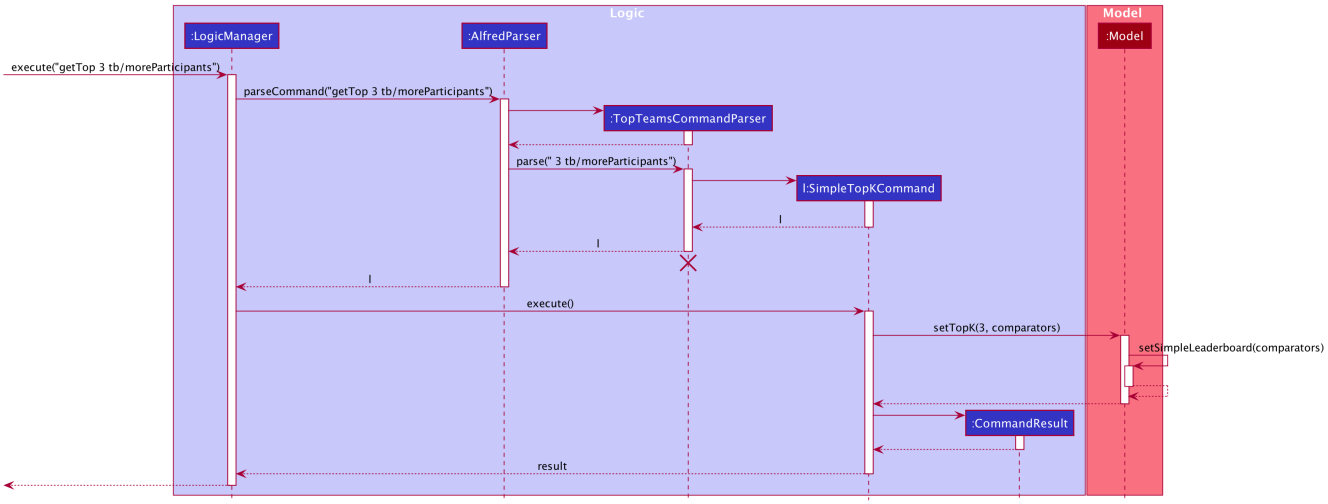


Figure 35. Interactions within Logic Component for getTop NUMBER Command

The above diagram follows a logic very similar to the `leaderboard` command's logic. However, in this case a `TopTeamsCommandParser` object is created to parse the arguments "3 tb/moreParticipants" which returns a `SimpleTopKCommand` object.

Moreover, the `SimpleTopCommand` object calls `Model` 's `setTopK(3, comparators)` method which essentially modifies the `sortedTeamList` within `Model` to only show the top three teams as per their scores and the relevant tiebreakers as per the list of comparators `comparators`.

It is also noteworthy that `Model` calls its own method `setSimpleLeaderboard(comparators)` which was associated with the `leaderboard` command. This is however a simple reuse of code to set the reset `Model` 's `sortedTeamList` and apply the relevant comparators to it, before the algorithm for fetching the top three (or any number) teams can be applied.

When it comes to the `getTop K` command being used with the "random" method of tiebreak, the flow of events is resembles the above very closely. Given below is the sequence diagram illustrating the flow of events which generates a result for the user when he types the command "getTop 3 tb/moreParticipants random". For your reference, here the prefix "tb/" is used to denote a tie-break method and "moreParticipants" is a tie-break method which gives a higher position to teams with more participants whereas "random" represents the "random" method of tiebreak.

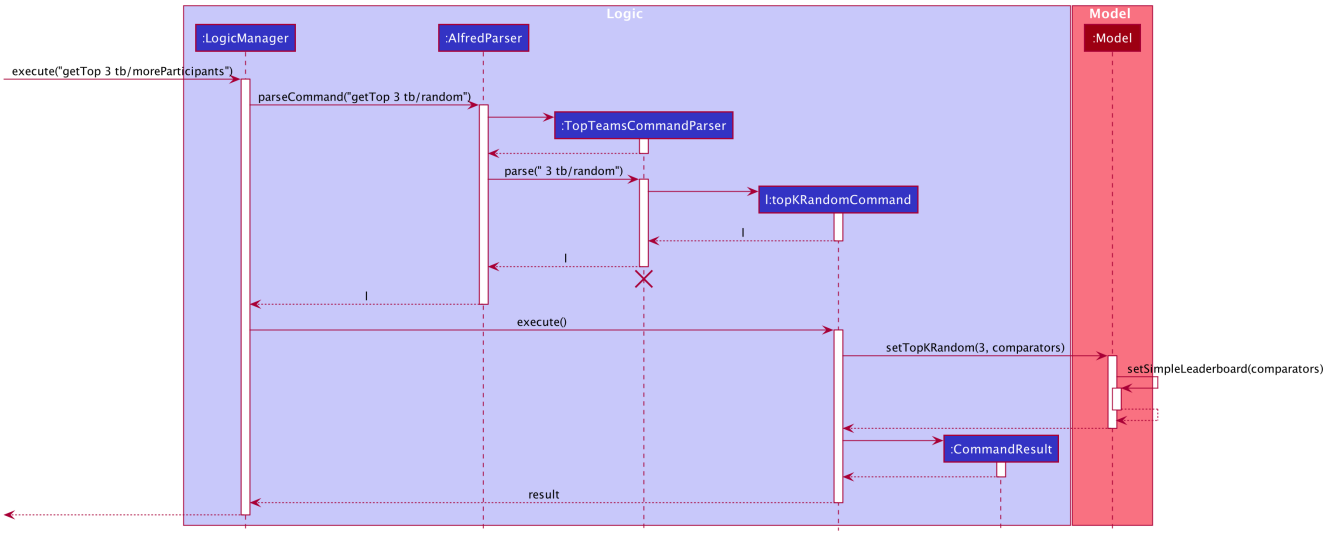


Figure 36. Interactions within Logic Component for getTop NUMBER with Random Winners

From the above diagram it can be inferred that the implementation of random winners does not deviate far from the implementation without random winners.

The first difference is that the `TopTeamsCommandParser` object now returns a `topKRandomCommand` object. Secondly, the `topKRandomCommand` object calls the `setTopKRandom(3, comparators)` method of `Mode`, which essentially modifies the `sortedTeamList` within `Model` to only show the top three teams as per their scores and the relevant tiebreakers as per the list of comparators `comparators`, and breaks any remaining ties based on the random method.

4.10.5. Design Consideration for Get Top Teams

Since the implementation of the `getTop K` command is almost identical to that of the `leaderboard` command, the design considerations made for the `leaderboard` command apply to the implementation of this feature as well (See Section 4.10.3, “Leaderboard Design Considerations”). However, there were some unique aspects we had to consider with regards to the `getTop K` command, all of which is detailed below.

Aspect: Where to Store the Top NUMBER Teams

- **Alternative 1 (Current Choice):** Use the `SortedList` in `ModelManager` used for the `leaderboard` command.
    - Pros: Requires lesser code - a new list would involve new getters and setters and additional code in the UI component to display this list on the UI.
    - Cons: Can be cause for confusion since `leaderboard` and `getTop K` commands would be using the same list.
- **Alternative 2:** Use a new `SortedList` object.
    - Pros: Less confusion as the `leaderboard` and `getTop K` commands use distinct lists for their operations.
    - Cons: Additional code and attention required to handle an additional list, which can lead to potential errors.

After careful consideration, "Alternative 1" was chosen as it would make fewer modification to `ModelManager` and the `UI` component, particularly with regards to adding duplicate code to handle the two different lists. Moreover, since the calls to `ModelManager` 's methods reset the `SortedList` storing the sorted teams, there is likely to be lesser confusion and room for error when handling a single list for the two different commands.

Aspect: Handling Situation when `K` is greater than the number of teams

- **Alternative 1 (Current Choice):** Show all the teams in the hackathon
    - Pros: Avoids frustrating the user if he constantly inputs a value greater than the number of teams, especially if he wants a quick overview.
    - Cons: Could be potentially seen as a bug as users and testers may notice a disparity between the number of teams shown and the number requested for.
- **Alternative 2:** Display an error to the user
    - Pros: May prevent some confusion in case user notices a disparity between the value he

inputted and the number of teams actually shown.

- Cons: Can be frustrating in case user wants a quick overview without having to worry about the total number of teams present.

We decided to prioritise user convenience in this situation and rather than displaying an error every time he inputs a value too large for `K` in the `getTop K` command, we decided to show all the teams. This aspect of the feature has been made abundantly clear in the User Guide and seeks to minimise user frustration especially since we do not want the user to worry too much about remembering how many teams have signed up.

Aspect: Implementation of Leaderboard

- **Alternative 1:** Implement `leaderboard` command with `getTop K` command keeping `K` as the size of the teamlist
  - Pros: Better and greater re-usage of code present within `ModelManager`.
  - Cons: Introduces coupling as changes made to the `getTop K` command will affect the `leaderboard` command.
- **Alternative 2 (Current Choice):** Have a separate method within `ModelManager` to handle the `leaderboard` command.
  - Pros: The two commands' logic are kept separate so neither affects the other in case of changes.
  - Cons: May be seen as a duplication of code.

The reason "Alternative 2" was selected was because `ModelManager` 's `setSimpleLeaderboard(ArrayList<Comparator<Team>> comparators)` method resets the `SortedList` of teams within `ModelManager` and applies the relevant comparators to it to sort it as desired. Hence, it is abstracting away this process into a single method so a better re-usage of code can take place in other methods. This method is reused by `ModelManager` 's `setTopK()` method when using the `getTop K` command, whereas the `setSimpleLeaderboard(ArrayList<Comparator<Team>> comparators)` method is sufficient to sort **all** the teams in the desired order. So instead of calling it again within another method, the `setSimpleLeaderboard(ArrayList<Comparator<Team>> comparators)` method which was meant to abstract away some processes is itself used to handle the `leaderboard` command. So indeed in the end, "Alternative 2" does not introduce duplication of code, but rather introduces better use of abstraction.

4.10.6. Tiebreaking and Filtering by Subject

In the above sections, the UML diagrams gloss over how tiebreaking and filtering by subjects is parsed and understood by Alfred. The basic of tie-breaking revolves around using `Comparator` s to sort the teams in a particular order. Each tiebreak method available in Alfred has a `Comparator` associated to it and all these can be found in the `Comparators` class.

The activity diagram below illustrates the internal workings within the `LeaderboardCommandParser` and `TopTeamsCommandParser` when parsing tiebreak methods. Do note that in the below diagram the term `Parser` encapsulates both of these `Parser` s as they operate in almost identical ways.

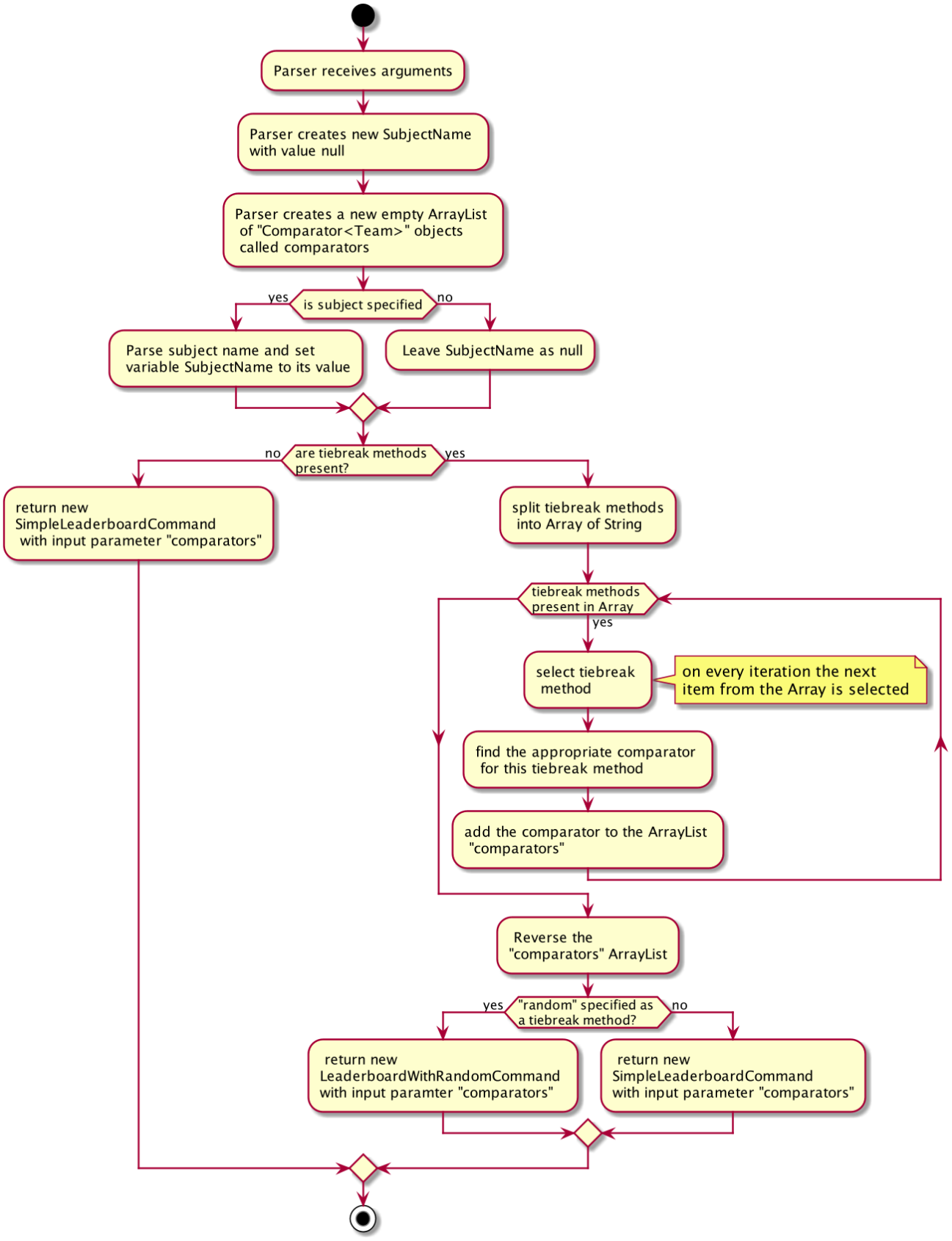


Figure 37. Handling of Tiebreak Methods

From the above there is one important aspect to note: the `ArrayList` of `Comparator<Team>` objects created in the second step. This is the `ArrayList` referred to in the previous sections which is tasked with containing and transferring the comparators which will be used to sort the teams in their appropriate order to form the leaderboard or top teams. This `ArrayList` is used as an input parameter for both `leaderboard` and `getTop K` command related classes, as explored in previous

sections.

Additionally, before `comparators` can be passed as an argument to the commands, it is reversed. This is done intentionally so the comparators related to each tiebreak method are applied in an order such that they preserve the oder the user wants the tiebreak methods to be applied.

Secondly, do note that the diagram above assumes there are no syntax errors made by the user when typing out the command. In case of any errors in the command, a `ParseException` would be thrown warning the user of such a situation. These have been excluded from the above diagram to prevent overcrowding and a deviation from the basic logic.

4.10.7. Design Considerations for Tiebreaking

Though this was a relatively straightforward subset of our `leaderboard` and `getTop K` command, there were still a few, small design considerations made.

Aspect: How to separate tiebreak methods

- **Alternative 1:** Each tiebreak method preceded by a new tiebreak prefix ("tb/")
  - Pros: Follows the paradigm followed by other commands.
  - Cons: Very tedious for the user to type and adds difficulty in parsing.
- **Alternative 2 (Current Choice):** Each tiebreak method separated by a single whitespace
  - Pros: Easier to implement and more convenient for the user to type as well. Code-wise it is easy and quick to customize which character to separate methods on.
  - Cons: Introduces a slight variation as other commands don't use whitespaces as separation methods which might confuse the user.

To better consider the user's needs and convenience, "Alternative 2" was selected. So as to not confuse users, this implementation has been made abundantly clear in the User Guide as well. From a developer's perspective, this implementation is also more customizable to better respond to changes in user's preferences as the separation character can easily be changes.

Aspect: Where to create comparators

- **Alternative 1:** Create comparators when parsing each tiebreak method
  - Pros: Easier to implement as it does not require additional classes.
  - Cons: Can overcrowd a single method or class especially as more comparison methods are added. This is also a poor use of abstraction and would not respect Single Responsibility Principle.
- **Alternative 2 (Current Choice):** Create a separate `Comparators` class where relevant comparators are created beforehand and can be invoke using static methods.
  - Pros: Better use of abstraction and maintains Single Responsibility Principle as the parser class can focus on solely parsing the user commands.

In the end, "Alternative 2" was selected as it follows better software engineering practices by making better use of abstraction. Despite requiring slightly more code and classes, it would still be

better than "Alternative 1" which breaks the Single Responsibility Principle as the parser class would have to be changed for changes in tiebreak methods in addition to changes in methods of parsing.

4.11. 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 4.12, “Configuration”)
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

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

4.12. Configuration

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

5. Documentation

Refer to the guide here.

6. Testing

Refer to the guide here.

7. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile:

- Human Resource Admin In-Charge of School of Computing 'Hackathon' Event
- has a need to manage a significant number of contacts
- has a need to register participants in bulk
- has a need to classify contacts into Mentor, Participants and Teams
- has a need to keep track of which member is in which Team
- has a need to keep track of the seating positions of each Team
- has a need to keep track of Mentor assignments to Teams
- has a need to keep track of the competition winners and prizes won
- has a need to search for specific Mentor, Team or Participant at times
- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition:

- manages different entities faster than a typical mouse/GUI driven app
- keeps track of the relationship between Participant, Team and Mentor, such that it can be referenced at times
- stores a significant number of entities in an organised, readable manner

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
* * *	Admin In-Charge	find a Entity by name	locate details of Entity without having to go through the entire list
* * *	Admin In-Charge	delete an Entity by name	remove entries I no longer need
* * *	Admin In-Charge	add an Entity by name and contact information	update the list of Entities
* * *	Admin In-Charge	updated an Entity by name and contact information	update the specific entries

Priority	As a …	I want to …	So that I can…
* * *	Admin In-Charge	register individuals en-masse(with provided registration information)	avoid tedious manual registration
* * *	Admin In-Charge	keep track of winning teams and the prizes won	ensure that the prize-giving ceremony runs smoothly
* * *	Admin In-Charge	keep track of winning teams and the prizes won	ensure that the prize-giving ceremony runs smoothly
* * *	Admin In-Charge	make sure that I will be notified on any wrong commands that I commandType	make sure that I do not accidentally clutter up my list of entries
* * *	Admin In-Charge	have a readable and organised User Interface	understand the output of my commands
* * *	Admin In-Charge	keep track of participants who signed up late or after the event has filled up into a waitlist	manage them in case available space turns up during the Event
* * *	Admin In-Charge	manually match Teams to Mentor	know which Mentor is in charge of a team
* * *	Admin In-Charge	keep track of where each Team or Mentor is seating	usher them to their places during the actual event
* *	Admin In-Charge	know my sponsor’s needs and arrival time	adequately cater to their needs and allocate manpower accordingly
* *	Admin In-Charge	keep track of inventory of swag	make sure they are adequately catered to all participants
* *	Admin In-Charge	keep track of amount of food or catering	make sure they are adequately catered to all participants
*	Admin In-Charge	automatically match Teams to Mentor by their expertise and project commandType of the Team	do not need to perform the matching manually

Priority	As a …	I want to …	So that I can…
*	Admin In-Charge	schedule meetings between Teams and Mentors	lets Mentors know when to consult each Team in an organised manner

{More to be added}

Appendix C: Use Cases

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

C.1. Use case: Delete an Entity Type (Participant, Mentor, Team)

MSS

1. User requests a list of an entity commandType
2. HackathonManager shows a list of that entity commandType
3. User requests to delete a specific entity in the list by name
4. HackathonManager deletes the person

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given name is invalid.

3a1. HackathonManager shows an error message.

Use case resumes at step 2.

C.2. Use case: Find an Entity of a specific Entity Type (Participant, Mentor, Team)

MSS

1. User requests a find an Entity of a specific Entity Type.
2. HackathonManager indicates success and shows the details of the Entity.

   Use case ends.

Extensions

1a. The Entity is not found in the list of Entities.

1a1. HackathonManager shows an error message.

Use case ends.

C.3. Use case: Create an Entity of a specific Entity Type(Participant, Mentor, Team)

MSS

1. User requests to create an Entity by specifying the Entity Type and contact information.
2. HackathonManager indicates success and shows the details of the Entity.

Use case ends.

C.4. Use case: Update an Entity of a specific Entity Type (Participant, Mentor, Team)

MSS

1. User requests a list of an entity commandType
2. HackathonManager shows a list of that entity commandType
3. User requests to update a specific entity in the list by name or index
4. HackathonManager updates the entity

Use case ends.

Extensions

1a. The name is not found it the list of Entities.

1a1. HackathonManager shows an error message.

1a2. User enters new name.

Steps 1a1-1a2 are repeated until the index or name is found in the list of Entities.

Use case resumes from step 4.

1b. The index is not found it the list of Entities.

1b1. HackathonManager shows an error message.

1b2. User enters new index.

Steps 1b1 - 1b2 are repeated until the index is found in the list of Entities.

Use case resumes from step 4.

C.5. Use case: Import external data through a CSV file

MSS

1. User writes a CSV file with Entity data.
2. User requests to import the CSV file located at user specified path into the HackathonManager.
3. HackathonManager finds and retrieves the CSV file.
4. HackathonManager adds each Entity in the CSV field.
5. HackathonManager displays original updated list of Entities.

   Use case ends.

Extensions

2a. User does not specify the path to the CSV file.

2a1. HackathonManager asks the user to specify the file path.

2a2. User specifies the file path.

Use case resumes from step 3.

2b. User also requests that the HackathonManager create an error file.

Use case resumes from step 3.

3a. HackathonManager cannot find the file or the path contains illegal characters (note that illegal path characters may vary from OS to OS).

3a1. HackathonManager informs user of failure of execution.

Use case ends.

4a. User specified CSV file contains invalid formatting

4a1. HackathonManager imports the valid lines only.

4a2. HackathonManager informs the user which lines were invalid and why.

Use case resumes from step 5 if user did not request for an error file.

4a3. HackathonManager creates a CSV file with the invalid lines at user specified path.

Use case resumes from step 5.

C.6. Use case: Export data to an external CSV file.

MSS

1. User requests that the HackathonManager export all of its data to an external CSV file at specified path.
2. HackathonManager exports its data to the file at specified path.
3. HackathonManager indicates success.

   Use case ends.

Extensions

1a. The user specifies which Entity Type data the HackathonManager should export.

   1a1. HackathonManager exports all of the data of the specified Entity Type to the file at specified path.

   Use case resumes from step 3.

1b. The user does not specify the path of the external file.

   1b1. The HackathonManager exports its data to the default file path.

   Use case resumes from step 3.

2a. The user specified path contains nonexistent directories and/or file.

   2a1. HackathonManager creates user specified directories and/or file.

   2a2. HackathonManager exports its data to the file at specified path.

   Use case resumes from step 3.

2b. The user specified path contains illegal characters (Note that illegal path characters may vary from OS to OS).

   2b1. HackathonManager notifies user of the failure of execution.

   Use case ends.

C.7. Use case: View the Leaderboard

MSS

1. User requests to see the Hackathon Leaderboard (that is the teams sorted in descending order of their points).
2. HackathonManager sorts the teams in descending order of their points.
3. HackathonManager displays the leaderboard.

Use case ends.

Extensions

2a. There are no teams currently registered

2a1. HackathonManager informs user that there are no teams to show leaderboard.

2a2. Use case ends.

C.8. Use case: View the Leaderboard with Tiebreaks

MSS

1. User requests to see the leaderboard and specifies tiebreak methods to break any ties.
2. HackathonManager sorts the team in descending order of their points.
3. HackathonManager breaks any ties between teams depending on the methods specified by the user.
4. HackathonManager displays the leaderboard.

Use case ends.

Extensions

2a. There are no teams currently registered

2a1. HackathonManager displays an error informing the user that there are no teams to show leaderboard.

Use case ends.

3a. No tiebreak methods specified by the user.

3a1. Use case resumes at step 4.

3b. Invalid tiebreak method inputted by user

3b1. Hackathon manager displays an error informing the user that the particular tiebreak method does not exist or is not supported.

3b2. User inputs the command again specifying tiebreak methods again.

3b3. Steps 3b1-3b2 are repeated until all tiebreak methods specified are correct.

Use case resumes from step 3.

C.9. Use case: Find the top scoring K Teams

MSS

1. User requests to see the Top K Teams in the Hackathon based on their score (highest score first), with K being the user input.

2. HackathonManager sorts the teams in descending order of their points and fetches the top K teams.
3. HackathonManager displays the top K teams with their respective scores.

   Use case ends.

Extensions

1a. The user input K as a negative, zero, invalid integer or non-integer value.

   1a1. HackathonManager shows an error message.

   1a2. User re-enters command with new user input of value K. Steps 1a1-1a2 are repeated until K is a correct value.

   Use case resumes from step 2.

2a. The user inputs K as a integer more than the available number of teams.

   2a1. HackathonManager fetches all the teams in descending order of their score.

   Use case resumes at step 3.

2b. There are no teams in the hackathon.

   2b1. HackathonManager shows an error message informing him there are no teams in the hackathon.

   Use case ends.

C.10. Use case: Find the top scoring K Teams with TieBreaks

MSS

1. User requests to see the Top K Teams in the Hackathon based on their score (highest score first), with K being the user input and specifies tie break methods to break any ties between teams with the same score.
2. HackathonManager sorts the teams in descending order of their points .
3. HackathonManager breaks any ties based on the methods specified by the user.
4. Hackathon manager fetches the top K teams.
5. HackathonManager displays the top K teams with their respective scores.

   Use case ends.

Extensions

1a. The user input K as a negative, zero, invalid integer or non-integer value.

   1a1. HackathonManager shows an error message.

1a2. User re-enters command with new user input of value K. Steps 1a1-1a2 are repeated until K is a correct value.

Use case resumes from step 2.

2a. There are no teams in the hackathon.

2a1. HackathonManager shows an error message informing him there are no teams in the hackathon.

Use case ends.

3a. No tiebreak methods specified by the user.

3a1. Use case resumes at step 4.

3b. Invalid tiebreak method inputted by user

3b1. Hackathon manager displays an error informing the user that the particular tiebreak method does not exist or is not supported.

3b2. User inputs the command again specifying tiebreak methods again.

3b3. Steps 3b1-3b2 are repeated until all tiebreak methods specified are correct.

Use case resumes from step 3.

4a. The user inputs K as a integer more than the available number of teams.

4a1. HackathonManager fetches all the teams in descending order of their score.

Use case resumes at step 5.

C.11. Use case: Find the ranking of all Teams

MSS

1. User requests for the top scorers of a specific category
2. HackathonManager shows the leaderboard of the category, with respective score of each team. Use case ends.

Extensions

1a. The category is not found.

1a1. HackathonManager shows an error message.

1a2. User enters category. Steps 1a1-1a2 are repeated until the category is found.

Use case resumes from step 2.

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. 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.
3. The system should not seem sluggish if it contains less than 1500 entities.
4. Project is not intended for use on mobile and only should be used on desktop.
5. The application assumes that the user is comfortable with the concept of the command line.
6. The application is meant to run offline.
7. The application is largely a personnel/HR manager, and is not expected to do anything more than that (eg hackathon finances etc).
8. The application is to be used for a single hackathon only and not for multiple hackathons.
9. The application assumes that the hackathon is a short term affair (no longer than 4 days).
10. The application assumes that this is an English medium hackathon and that no non-English names are expected.
11. The GUI should display the result of commands in an intuitive, organized manner that is readable by the laymen(as part of the organization/ affordability of the application).

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Private contact detail

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

Logging

Logging uses file(s) containing information about the activity of a computer program for the developers to consult and monitor.

Entity

Entities are the main objects Alfred stores. The Entities are Participant, Mentor and Team as described below.

Participant

It represents a participant taking part in the hackathon

Mentor

It represents a mentor available for teams to choose

Team

Team is the base unit of this project. It contains references to an associated list of participants

and an optional mentor.

Appendix F: Product Survey

Google Sheets

Author: Google

Pros:

- This is extremely versatile as Google Sheets come with a list of extremely helpful macros that could help in the storage of participants.
- The display and UI of Google Sheets is extremely intuitive and will come as second nature to anyone using the web.
- Convenient and accessible by multiple HR personnel simultaneously.

Cons:

- Google Sheets has no concept of objects and thus it cannot accurately depict the relationships between our different entities.
- As above, it is hard to look for relationships between our entities, such as Team/Participant associations.
- Google Sheets may be useful for storing information, but it does not support command line arguments.
- Google Sheets is also unable to perform input validation as it lacks the logic to do so.

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.
      Note: If you are a OS X user, you might need to run this from your command line instead.
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 Participant

1. Deleting a Participant while all participants are listed

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

   b. Test case: `delete participant P-1`
   Expected: Participant with id P-1. Details of the deleted contact shown in the status message. Timestamp in the status bar is updated.

   c. Test case: `delete participant P-101212323`
   Expected: No participant 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.

G.3. Saving data

1. Dealing with missing/corrupted data files

   a. Prerequisites: Create a JSON with corrupted data, or any data at all

   b. Test case: Start the application. Logger should kindly inform you that the storage files are corrupted and hence it defaults to using empty lists.