

Abhiman Yadav - Project Portfolio

PROJECT: Alfred - The Hackathon Butler

Overview

For our software engineering project, my team of five software engineering students, including myself, were tasked with either improving an existing addressbook application's codebase, or completely modifying it into a new application. We opted to morph the addressbook codebase into a command-line based hackathon organizing tool which we named "Alfred". Alfred is a desktop application targeted towards hackathon organizers and intends to aid them with several organizational and logistical needs, such as tracking the participants, teams and mentors involved in the hackathon; managing the relations between different these entities; and judging and determining winners within the hackathon. All of this intends to package the various complex tools required to organise a hackathon into a single desktop application, resembling the image below.

Figure 1. Alfred's UI

My main role in this project was to design and write code for the judging and leaderboard functions we intended to equip our application with. Judging and leaderboard are essential components of a hackathon as every competition is bound to have winners based on their scores. The following sections of this document highlight in more detail the features I implemented and the enhancements I added to this project. Additionally, it also explores the relevant documentation I added to our application user guide and developer guide with regards to the enhancements I made.

Summary of contributions

Major enhancement: added the ability to assign scores to teams

- ¥ What it does: allows the user to manage teams' scores by providing functionalities to add, subtract, set and reset the scores of the teams present within the hackathon.
- ¥ Justification: Scoring is one of the core processes which take place within a hackathon in order to determine winners. This feature allows the user to conveniently change a team's score depending on their needs.
- ¥ Highlights: This feature is equipped with appropriate feedback messages and error detection capabilities to ensure that the user uses it correctly to the fullest of its capabilities without leading to errors. The implementation was challenging as there were several edge cases and exceptional cases which needed to be considered. Thorough attention also had to be paid to make the best use of abstraction and polymorphism while implementing this feature to minimise the amount of duplicate code and ensure adherence to ideal software engineering practices.

Major enhancement: implemented the ability to view the leaderboard and top teams within the hackathon

- ¥ What it does: allows the user to see the hackathon's leaderboard or any number of top teams within the hackathon to determine winners for the competition. Additionally, it provides tiebreaking capabilities to allow the user to break ties between teams with equal scores, based on certain metrics. Lastly, it allows the user to determine winners by categories by allowing the user to filter the leaderboard or top teams by the category they belong in.
- ¥ Justification: At the end of a hackathon, the organizers need to determine the winners of the hackathon. This feature allows them to easily do so easily by running simple single line commands, rather than having to scroll and squint at different teams in the hackathon.
- ¥ Highlights: This enhancement was particularly difficult to implement as it had several sub-parts to it, especially in handling cases when teams are ties. This required me to devise and code few complicated algorithms to ensure that the multiple edge cases, such as when teams are still tied after applying tiebreak methods, are appropriately handled in order to display the correct results to the user.

Code Contributed: Please use the following link [Code_Contribution](#) to view the code I contributed towards this project on the contribution analysis tool RepoSense. You may also use the available filters to view my functional code and test code separately.

Other contributions:

- ¥ Project management:
 - ! Managed our team repository's Issue Tracker and Milestones Tracker.
 - ! Helped maintain good level code quality by providing comments on my teammates' codes.
- ¥ Enhancements to existing features:
 - ! Refactored the addressbook application's original parser classes to better suit the needs of

our application (Pull requests [#62](#), [#98](#), [#127](#)).

! Deprecated addressbook code to remove it from the codebase (Pull request [#161](#))

! Wrote tests for the Parser classes to ensure they functioned correctly. (Pull requests [#130](#))

! I abstracted several common bits of code into single methods which were then used by my teammates (Pull requests [#298](#)).

¥ Documentation:

! Wrote the Score, Leaderboard and Top Teams command sections of the user guide.

! Explained the implementation of the Score, Leaderboard and Top Teams Command in the developer guide.

! Made cosmetic changes to the contents of the Developer Guide to suit our application (Pull requests [#154](#)).

Contributions to the User Guide

Alfred's user guide is a thorough and intuitive guide which explains to its users how to properly use each of its features. My extensive contributions to it showcase my ability to write documentation targeting the end-users. This section contains an excerpt from my additions to the user guide with regards to my Leaderboard and Get Top Teams feature which explains how to use the `leaderboard` command on its own and with tiebreaking.

Do note the `getTop k` command which you may see below is another feature I implemented but is not part of this excerpt to make space for more content. You may also view the full, uncropped version of my contributions in Alfred's User Guide found [here](#).

Team Rankings: `leaderboard` and `getTop k`

In addition to assigning scores to teams, Alfred also facilitates viewing the leaderboard and fetching the top teams in the hackathon with ease. The following subsections explain how to do so within Alfred.

View Leaderboard: `leaderboard`

Displays the ranking of all the teams in the hackathon in descending order of their points.

- ¥ Once you run this command, Alfred's UI will display a list of all the teams stored within Alfred sorted in descending order of their points.
- ¥ By default Alfred sorts teams with equal points in the order they were added into Alfred, based on their ID.

Format: `leaderboard`

Running `leaderboard` will show the following on the GUI:

Figure 2. GUI Display for Command "leaderboard"

Extensions to `leaderboard` and `getTop K` Command

To provide additional functionalities to the `leaderboard` and `getTop k` commands, there are few extensions that can be added to these two commands to allow you to customize them to your needs. These extensions are explored below.

Tie-Break

By default Alfred `leaderboard` and `getTop k` commands fetch and display teams in descending order of their score, and by the order they were added into Alfred in case of tied scores.

Alfred's tiebreak feature provides an extension to the `leaderboard` and `getTop k` commands allowing you to choose how you want to break the tie between the teams when calling these commands. To break a tie, follow the following format:

¥ `leaderboard tb/METHOD_1 METHOD_2 METHOD_3` in the case of a `leaderboard` command

¥ `getTop NUMBER tb/METHOD_1 METHOD_2 METHOD_3` in the case of a `getTop NUMBER` command

where `METHOD_N` is one of the following currently available tie-break methods:

¥ `moreParticipants`: teams with more participants are win the tie.

¥ `lessParticipants`: teams with lesser participants are win the tie.

¥ `higherId`: teams registered more recently (hence the highest ID) win the tie.

¥ `lowerId`: teams registered earlier (hence the lowest ID) win the tie.

¥ `random`: in case all methods used yield no distinct winner, `random` can be used as a method of last resort to break a tie in favour of a randomly chosen team.

- ¥ You may choose one or more methods from the above list to break the tie. You need to precede the tie-break methods with the prefix `tb/` and separate each method with a single space for Alfred to properly understand them.
- ¥ Use the prefix `"tb/"` with discretion as Alfred will only select tiebreak methods followed by the last `"tb/"` prefix if more than one such prefix is specified in the command.
- ¥ Do note that the tie-break methods will be applied in the order in which you state them. That is, first `METHOD_1` will be applied to break the ties, and only then will `METHOD_2` be applied to break any remaining ties, if the command `leaderboard tb/METHOD_1 METHOD_2` is called.
- ¥ The `getTop NUMBER` command may still display teams more than the value of `NUMBER` if Alfred was still unsuccessful in breaking certain ties despite applying the tie-break methods you stated.
- ¥ When using the `random` method, it must be the last stated tie-break method if it is being used alongside other tie-break methods.

Example:

- ¥ `leaderboard tb/moreParticipants lowerId` will display the leaderboard on the UI with Alfred breaking the tie between teams with equal scores based on which team has more participants, and if the number of participants is equal then by which team has the lower ID.

Figure 3. GUI Display for Command "leaderboard tb/moreParticipants lowerId"

Note that in the above team "BroBro" comes above team "Bro" despite having the same number of points, as "BroBro" has more participants. Secondly, "Amazon Warriors" comes first before "Teen Titans" despite having the same score and number of participants, since "Amazon Warriors" has a lower ID.

Contributions to the Developer Guide

Alfred's developer guide provides an in-depth look into the architecture of how Alfred and its various features were implemented. My extensive contributions to this document exhibit my ability to write technical documentation and showcase the technical depth of my contribution to the project. This section contains an excerpt from the developer guide which shows the contribution I made with regards to my Leaderboard feature. It explains to the reader how the `leaderboard` command has been implemented and design considerations which were made when implementing it. You may also view the full version of my contributions in Alfred's Developer Guide found [here](#).

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.

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 `Model Manager` 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.
- ¥ use a `SubjectName` object to filter the leaderboard or top teams by a certain category, if specified by the user.

The class diagram below provides a high level representation of the Object-Oriented solution devised to implement the `leaderboard` and `getTop K` commands.

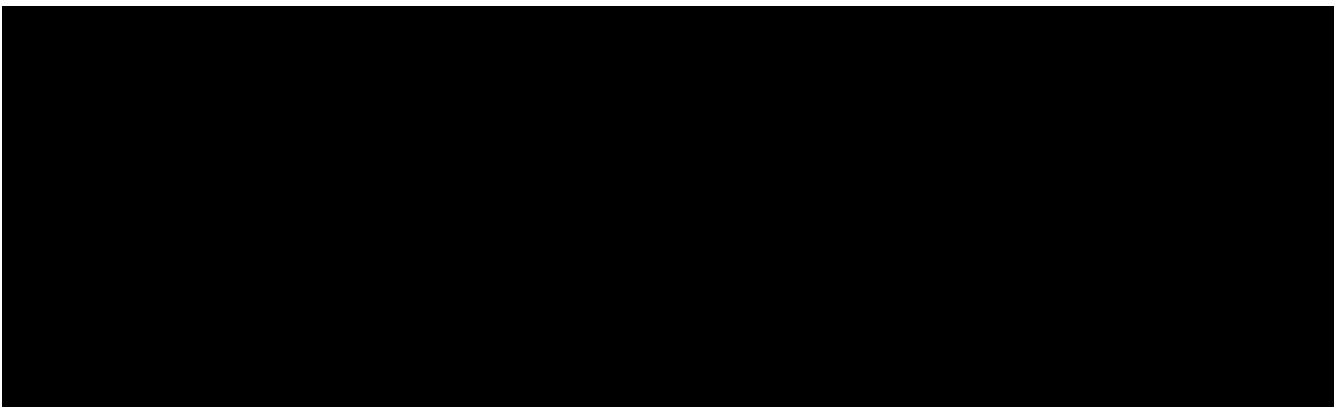


Figure 4. Leaderboard and Get Top Teams Implementation Overview

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 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.

Leaderboard Command Implementation

The `leaderboard` command fetches a leaderboard consisting of all the teams registered for the hackathon, in descending order of their score. Moreover, if the user specifies a `SubjectName` then the leaderboard will only consist of teams with that particular subject.

Additionally, if tiebreak methods are specified, ties between the teams will be broken in one of two ways (or a combination of both):

- ¥ 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 s/Social`. For your reference, here the prefix "tb/" is used to precede a tie-break method, "moreParticipants" is a tie-break method which gives a higher position to teams with more participants, and "Social" is a `SubjectName` within Alfred. Essentially this demonstrates the flow for a "Comparison-based tiebreak".

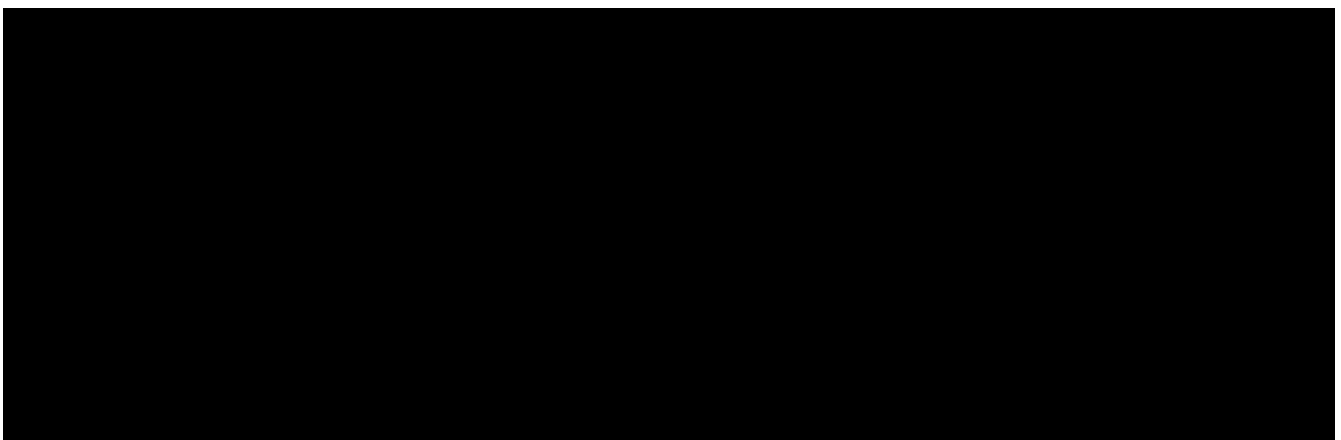


Figure 5. Interactions within Logic Component for SimpleLeaderboardCommand

The observations of the above diagram can be put into the following steps:

- ¥ Step 1: `Logi cManager` starts executing the user's command and calls the `AI fredParser` to parse it.
- ¥ Step 2: `AI fredParser` find the appropriate `Parser` to parse the command and creates a new `LeaderboardCommandParser` to parse the arguments of the leaderboard command, essentially "tb/moreParticipants s/Social".
- ¥ Step 3: The `LeaderboardCommandParser` then parses the arguments and is responsible for:
 - ! Checking whether the user has specified a subject. If so it sets the value of a local variable "subjectName" of type "SubjectName" to the appropriate subject, otherwise it stays as null.
 - ! Checking whether any tiebreak methods are present.
 - ! Parsing 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.
- ¥ Step 4: `LeaderboardCommandParser` then creates a new `SimpleLeaderboardCommand` object with the above list of comparators and subject as input parameters. This is then returned all the way to `Logi cManager`
- ¥ Step 5: `Logi cManager` then executes the `SimpleLeaderboardCommand` object upon which the `SimpleLeaderboardCommand` object calls `Model 's setSimpleLeaderboard(comparators, subjectName)` where comparators is the `ArrayList` of comparators and subjectName is the `SubjectName` used to create the `SimpleLeaderboardCommand`.
- ¥ Step 6: `Model 's setSimpleLeaderboard(comparators, subjectName)` method updates the `sortedTeamList` within `Model` itself, by applying the comparators to it and filtering the list by the subjectName specified.
- ¥ Step 7: Upon doing so, the `SimpleLeaderboardCommand` 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 and signals the UI to display the teams from the `sortedTeamList`.

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 s/Social` then the `SimpleLeaderboardCommand` object would be created with an empty `ArrayList` of comparators. If the user's input did not specify any subject, hence just being `Leaderboard`, then the `SimpleLeaderboardCommand` object would be created with the `SubjectName` variable "subjectName" being null, in which case no filtering of `sortedTeamList` takes place. The flow of events for this particular scenario would be unchanged from the above illustration.

The `Leaderboard` command with the tiebreak method `random` follows a slightly different sequence. Given below is the sequence diagram illustrating the flow of events when the user 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 6. 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.

However, in this case the `LeaderboardWithRandomCommand` class calls the `setTopK(teamListSize, comparators, subjectName)` method of `Model` which essentially filters out the teams with subject "subjectName", 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`, thereby reflecting the total number of teams in the hackathon.

Secondly, `Model` calls its own method `setSimpleLeaderboard(comparators, subjectName)` which was used for the `leaderboard` command without random tiebreak. This method abstracts the process of clearing `sortedTeamList` of any sorting, filters it by `SubjectName` if required, and then applies the new comparators to it. It is used to fetch and appropriately sort the appropriate teams in `sortedTeamList` before the algorithm for random winners can be applied to the `sortedTeamList`.

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 features' 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, particularly with the convenience of sorting it allows through its API, 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.