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 decided to opt for the latter of the two options and 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 application.

Summary of contributions

- **Major enhancement**: added **the ability to undo/redo previous commands**
  - What it does: allows the user to undo all previous commands one at a time. Preceding undo commands can be reversed by using the redo command.
  - Justification: This feature improves the product significantly because a user can make mistakes in commands and the app should provide a convenient way to rectify them.
  - Highlights: This enhancement affects existing commands and commands to be added in future. It required an in-depth analysis of design alternatives. The implementation too was challenging as it required changes to existing commands.
  - Credits: *{mention here if you reused any code/ideas from elsewhere or if a third-party library is heavily used in the feature so that a reader can make a more accurate judgement of how much effort went into the feature}*
- **Minor enhancement**: added a history command that allows the user to navigate to previous commands using up/down keys.
- **Code contributed**: [Functional code] [Test code] *{give links to collated code files}*
- **Other contributions**:
  - Project management:
    - Managed releases `v1.3` - `v1.5rc` (3 releases) on GitHub
  - Enhancements to existing features:
    - Updated the GUI color scheme (Pull requests #33, #34)

        - Wrote additional tests for existing features to increase coverage from 88% to 92% (Pull requests #36, #38)
    - Documentation:
        - Did cosmetic tweaks to existing contents of the User Guide: #14
    - Community:
        - PRs reviewed (with non-trivial review comments): #12, #32, #19, #42
        - Contributed to forum discussions (examples: 1, 2, 3, 4)
        - Reported bugs and suggestions for other teams in the class (examples: 1, 2, 3)
        - Some parts of the history feature I added was adopted by several other class mates (1, 2)
    - Tools:
        - Integrated a third party library (Natty) to the project (#42)
        - Integrated a new Github plugin (CircleCI) to the team repo

{you can add/remove categories in the list above}

Contributions to the User Guide

Given below are sections I contributed to the User Guide. They showcase my ability to write documentation targeting end-users.

Contributions to the Developer Guide

Given below are sections I contributed to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.

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 `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.

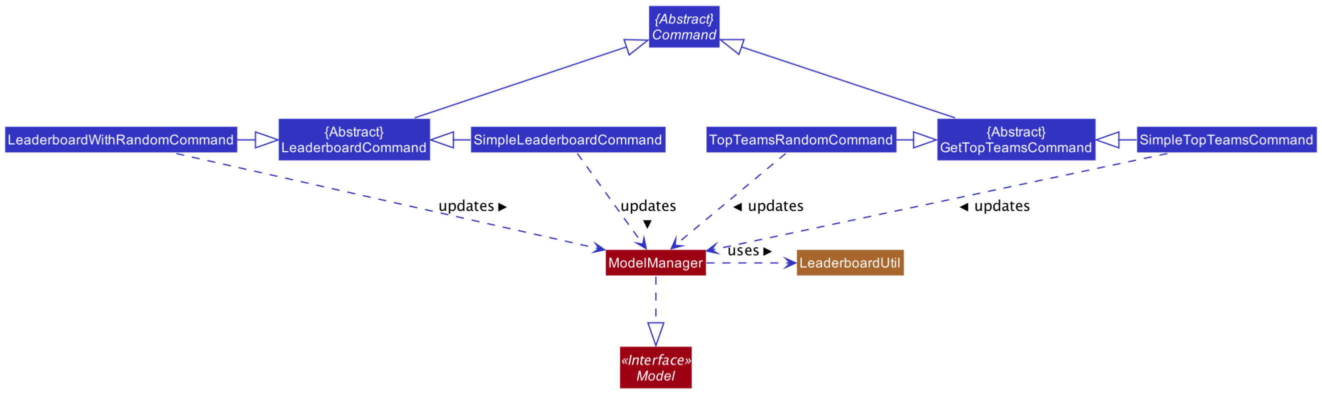

Figure 1. Leaderboard and Get Top Teams Implementation Overview

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.

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".

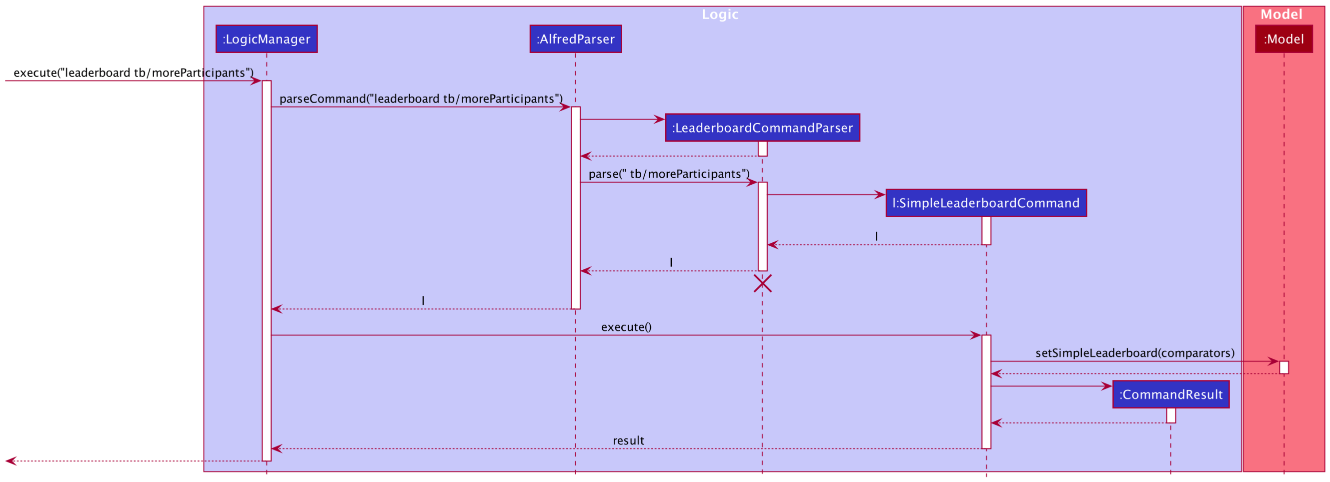
Logic
Model
:LogicManager
:AlfredParser
:LeaderboardCommandParser
l:SimpleLeaderboardCommand
:Model
:CommandResult
execute("leaderboard tb/moreParticipants")
parseCommand("leaderboard tb/moreParticipants")
parse(" tb/moreParticipants")
l
l
l
execute()
setSimpleLeaderboard(comparators)
result

Figure 2. 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.

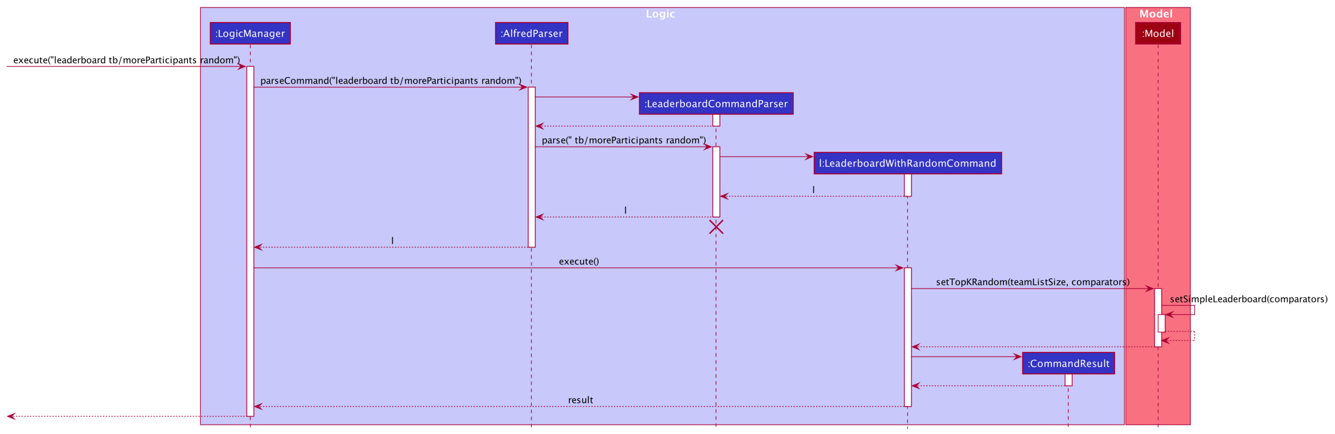
Logic
Model
:LogicManager
:AlfredParser
:Model
execute("leaderboard tb/moreParticipants random")
parseCommand("leaderboard tb/moreParticipants random")
:LeaderboardCommandParser
parse(" tb/moreParticipants random")
l:LeaderboardWithRandomCommand
l
l
l
execute()
setTopKRandom(teamListSize, comparators)
setSimpleLeaderboard(comparators)
:CommandResult
result

Figure 3. 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`.

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.

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.

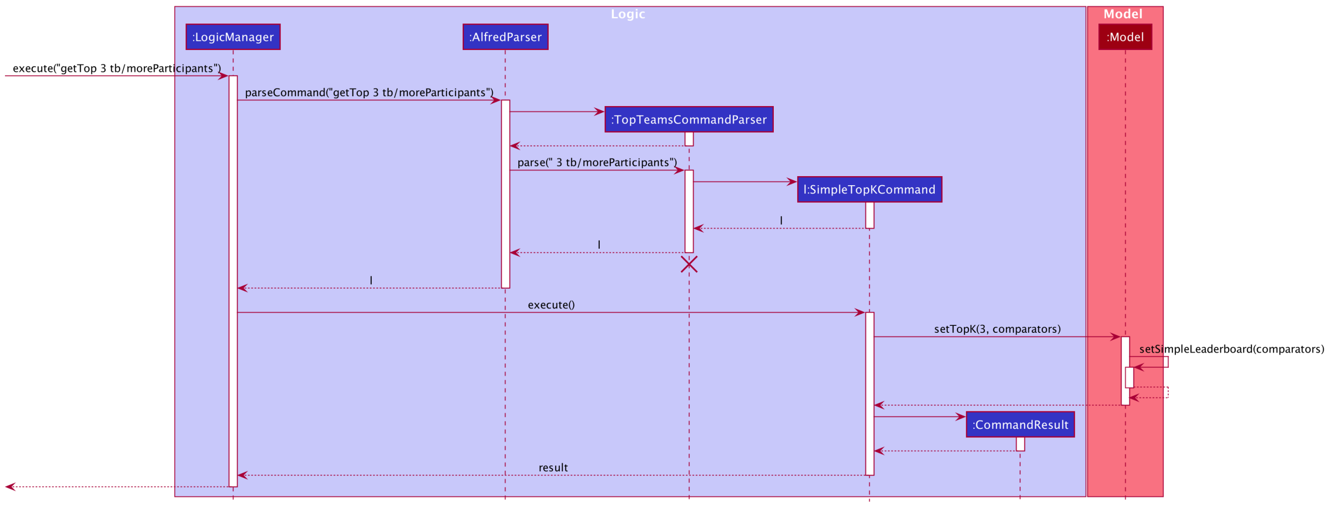
Logic
Model
:LogicManager
:AlfredParser
:Model
execute("getTop 3 tb/moreParticipants")
parseCommand("getTop 3 tb/moreParticipants")
:TopTeamsCommandParser
parse(" 3 tb/moreParticipants")
l:SimpleTopKCommand
l
l
l
execute()
setTopK(3, comparators)
setSimpleLeaderboard(comparators)
:CommandResult
result

Figure 4. 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.

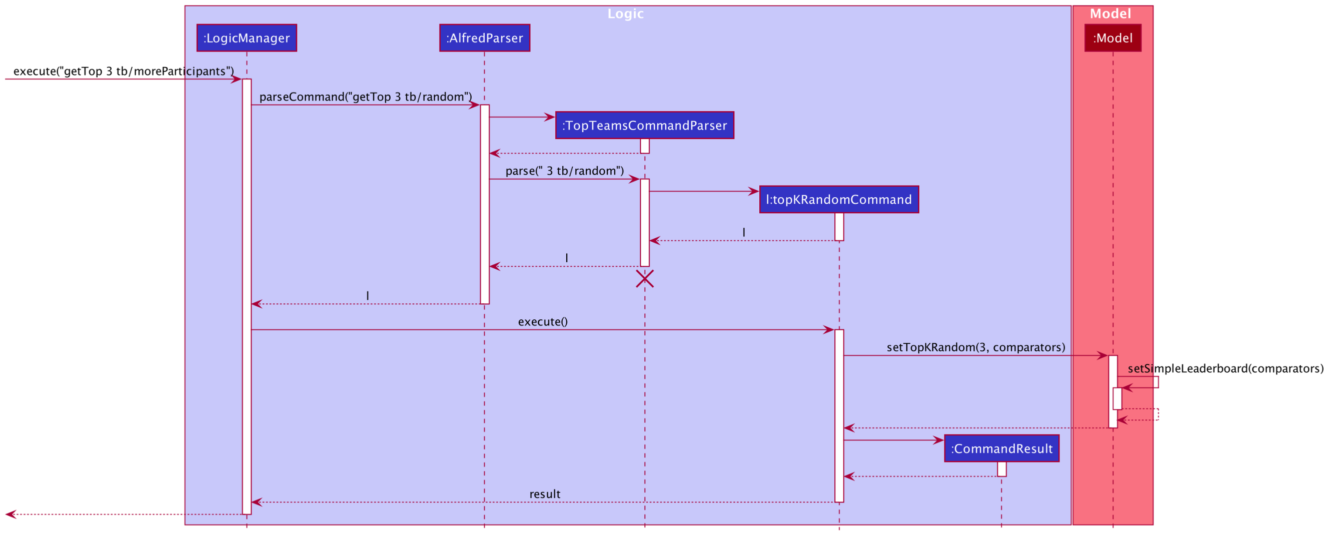

Figure 5. Interactions within Logic Component for getTop NUMBER with Random Winners

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.

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 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.

Tiebreaking

In the above sections, the UML diagrams gloss over how tiebreaking is really 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.

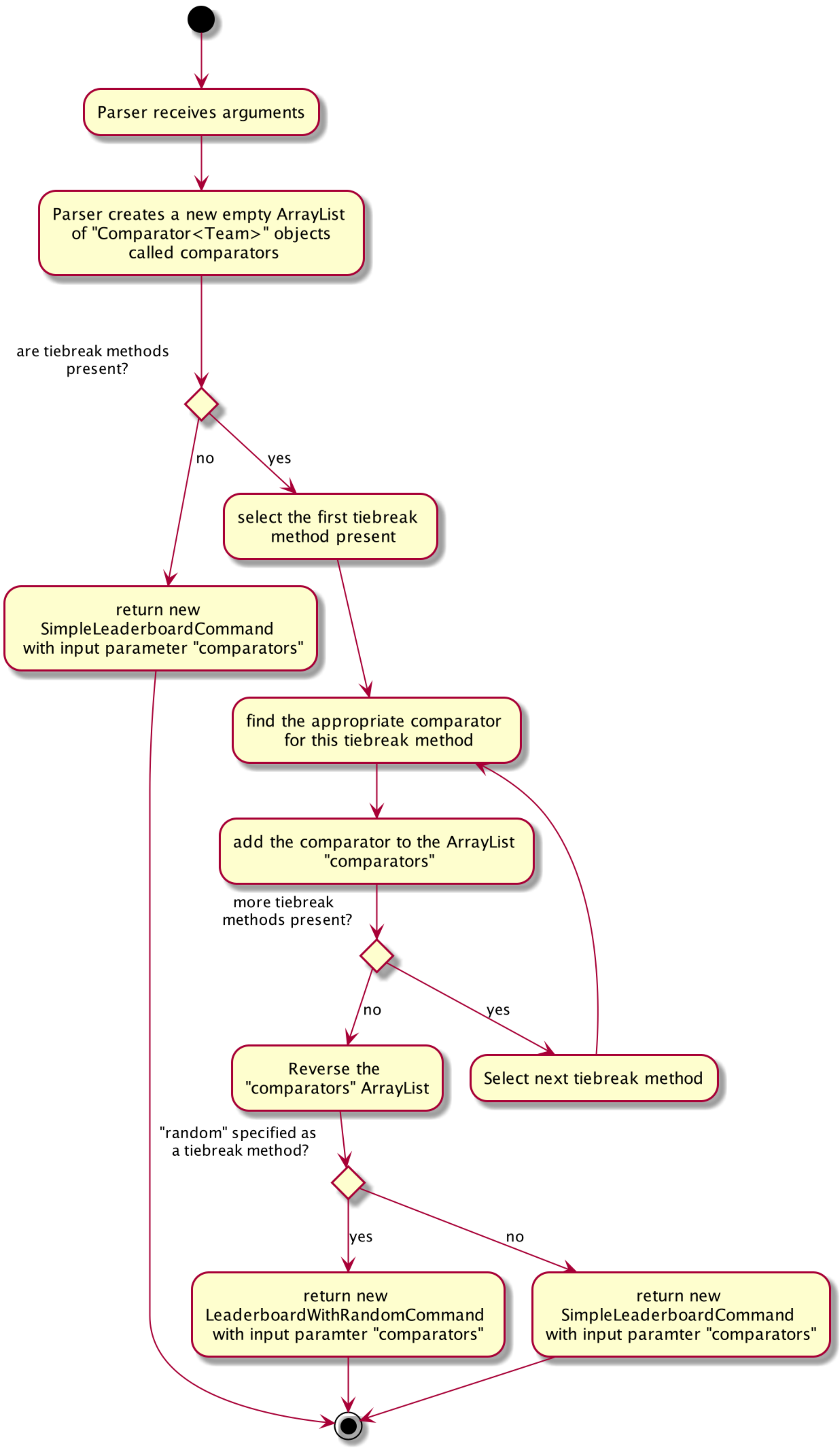
Parser receives arguments
Parser creates a new empty ArrayList of "Comparator<Team>" objects called comparators
are tiebreak methods present?
no
yes
select the first tiebreak method present
return new SimpleLeaderboardCommand with input parameter "comparators"
find the appropriate comparator for this tiebreak method
add the comparator to the ArrayList "comparators"
more tiebreak methods present?
no
yes
Reverse the "comparators" ArrayList
Select next tiebreak method
"random" specified as a tiebreak method?
yes
no
return new LeaderboardWithRandomCommand with input paramter "comparators"
return new SimpleLeaderboardCommand with input paramter "comparators"

Figure 6. 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 intently 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.

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.

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 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

Configuration

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

Documentation

Refer to the guide here.

Testing

Refer to the guide here.

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

Priority	As a …	I want to …	So that I can…
* * *	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
* * *	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

Priority	As a …	I want to …	So that I can…
* *	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
*	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)

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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 … }

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.

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.

PROJECT: PowerPointLabs

{Optionally, you may include other projects in your portfolio.}