# Brian Yen - Project Portfolio

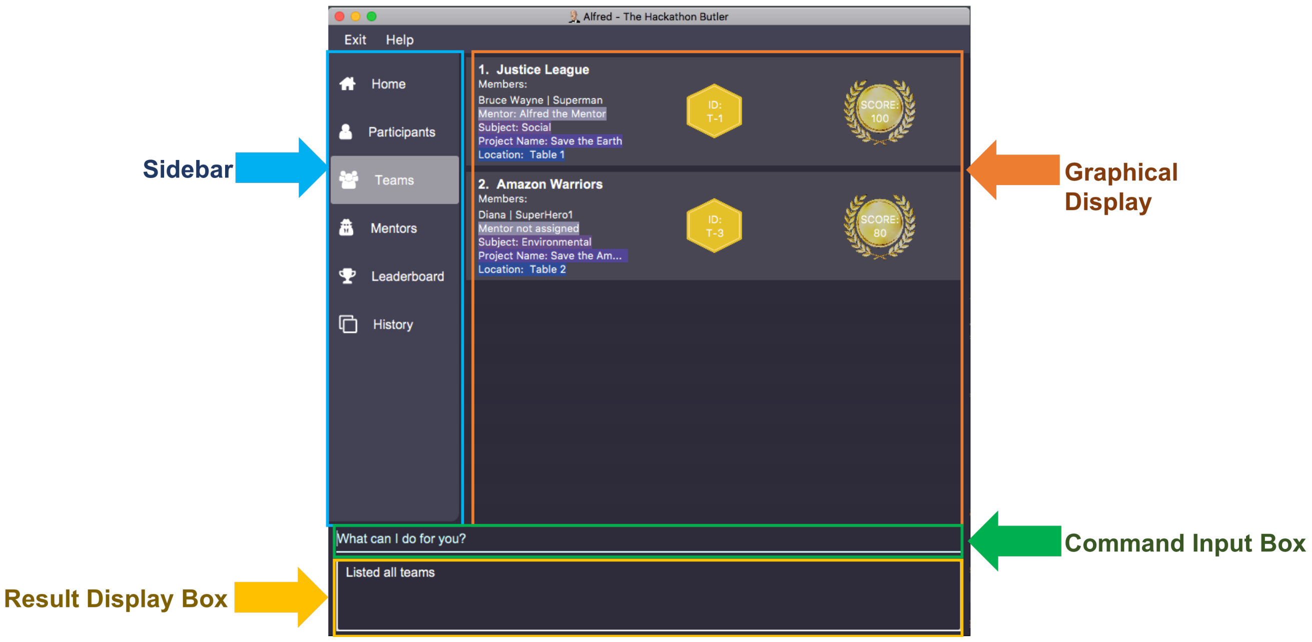
## PROJECT: Alfred - The Hackathon Butler

---

## Overview

As part of a Software Engineering project, my team and I were tasked with transforming an existing AddressBook desktop application into any application of our choice, so long as it maintained a Command Line Interface and had a Graphical Display. We chose to transform the existing application into a Hackathon manager that is capable of managing the different workflows required for any Human Resource/Administrative Manager to organise a Hackathon. Named after Batman’s trusty butler, Alfred allows Hackathon organisers to keep track of the individuals taking part in the Hackathon, assign participants to teams, visualise statistics and even import data from .csv files.

After several rounds of iterations and improvements, we are proud to present Alfred:



1. Justice League
Members:
Bruce Wayne | Superman
Mentor: Alfred the Mentor
Subject: Social
Project Name: Save the Earth
Location: Table 1

2. Amazon Warriors
Members:
Diana | SuperHero1
Mentor not assigned
Subject: Environmental
Project Name: Save the Am...
Location: Table 2

Listed all teams

*Figure 1. Alfred’s User Interface*

My main contribution to the team was to adapt the existing Storage implementation to fit the new application’s purpose, and to implement the `undo`, `redo` and `history` features. In addition to this, I also implemented the Command History Navigation feature, which allows users to navigate to previously executed commands using keyboard shortcuts - a near-universal convenience feature provided in several Command Line prompts.

## Summary of contributions

## Features Contributions

- **Major Enhancement** Adapted Storage to fit the needs of Alfred.
    - Purpose: My contribution was to allow Alfred’s data types to be stored to and read from disc. I also ensured that Alfred’s various EntityLists could be serialized to JSON and saved to disc in a convenient manner, while guaranteeing that all data read from disc is first validated before being sent to Alfred so that corrupted data will not cause further problems upstream.
    - Justification: Storage is a key component in Alfred and ensures that all data changes can be automatically saved and persisted. This is critical for Alfred. Hackathon-organising is a long-term, large-scale effort, so being able to save your data reliably is key to the application’s success.
    - Highlights: There is no need for users to save their data as Alfred always automatically saves the data to Storage.
- **Major Enhancement** Created an undo/redo feature.
    - Purpose: Allows users to undo/redo previously executed commands. This allows users to easily correct mistakes while using Alfred.
    - Justification: Should users make a mistake or have a need to reverse previously executed commands, this feature allows them to quickly revert changes by invoking the `undo` command, improving their overall productivity and streamlining workflows with the application. Should they change their mind and actually wish to execute the command again, there is no need for them to re-type and execute the command, as they can simply invoke the `redo` command.
    - Highlights: You can undo/redo multiple commands at once, by invoking `undo N`/`redo N`, where N is an integer.
- **Minor Enhancement** Allow users to view the history of previously executed commands
    - Purpose: Allows users to keep track of the changes they make
    - Justification: Especially after invoking several commands, it can be difficult for users to remember the order of commands they had previously executed. Hence, the `history` command provides a convenient visualisation of previously executed commands to better inform users on which commands they would like to `undo`/`redo`.
- **Minor Enhancement** Allow users to navigate to previously executed commands using the ALT + UP/DOWN Arrow Keys
    - Purpose: Allow quick access to previously executed commands, as opposed to re-typing everything.
    - Justification: This is a convenience feature that most Command Line prompts have, so it is a nice feature for Alfred to include, given that Alfred is ultimately a Command Line Interface.
    - Highlights: Users can navigate up to 50 previous commands.

## Code Contributions

These are some of the code I have written:

- Functional Code
- Test Code

## Other Contributions

These are some of the other ways I have helped my team along the way to develop Alfred.

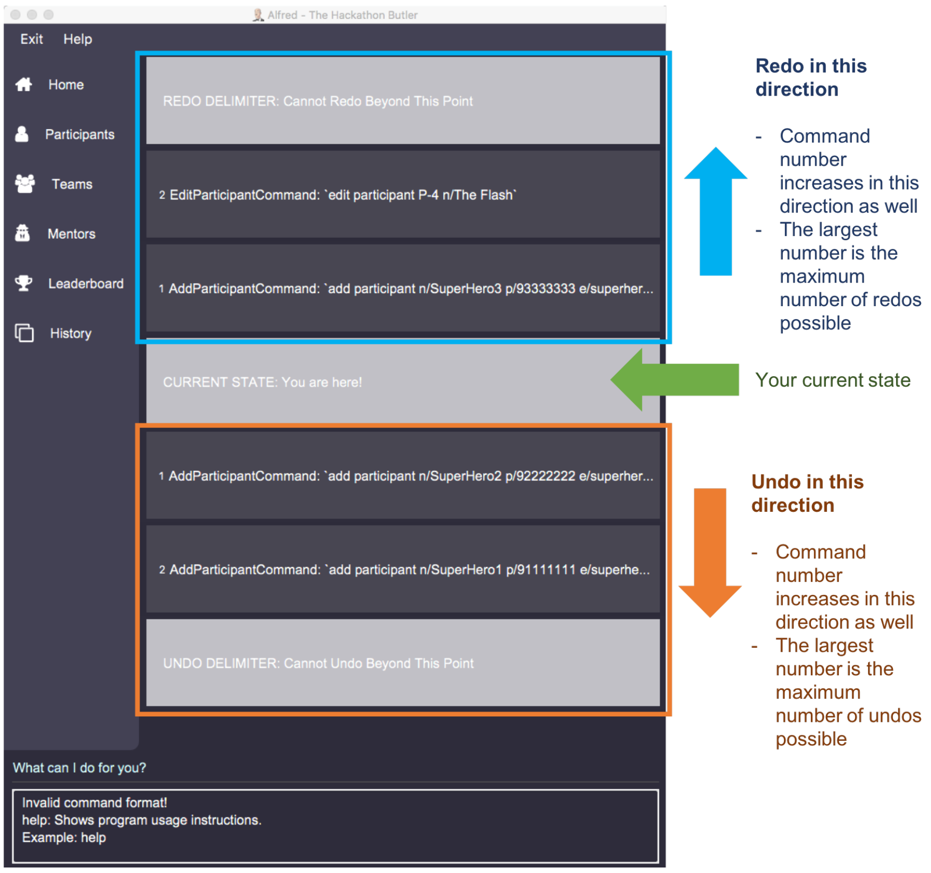
- Project management:
  - Helped manage releases `v1.2` - `v1.4` on GitHub
- Tests
  - Helped write tests to ensure that test code coverage is maintained at a healthy level
  - These tests helped ensure that the implementation of new features by my teammates can be verified to not break existing features, therebys streamlining Alfred's development.
- Documentation:
  - In charge of team's Documentation.
  - Helped standardise the format for both User Guide and Developer Guide
  - Helped proof-read the User Guide and Developer Guide and inform teammates of any areas that require changes, with the readability of the documentation being a top priority.
- Community:
  - reviewed PRs (with non-trivial review comments)
  - participated actively in discussions with my teammates for their features

# 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. The following text is taken from the History feature section of the User Guide, and explains how to use the History feature. For more information on the features I implemented, please feel free to look through the User Guide here.*

## Show Command History: `history`

Shows you up to the last 50 commands that you executed.


Alfred - The Hackathon Butler
Exit
Help
Home
Participants
Teams
Mentors
Leaderboard
History
Redo in this direction
- Command number increases in this direction as well
- The largest number is the maximum number of redos possible
Your current state
Undo in this direction
- Command number increases in this direction as well
- The largest number is the maximum number of undos possible

REDO DELIMITER: Cannot Redo Beyond This Point

2 EditParticipantCommand: `edit participant P-4 n/The Flash`

1 AddParticipantCommand: `add participant n/SuperHero3 p/93333333 e/superher...

CURRENT STATE: You are here!

1 AddParticipantCommand: `add participant n/SuperHero2 p/92222222 e/superher...

2 AddParticipantCommand: `add participant n/SuperHero1 p/91111111 e/superhe...

UNDO DELIMITER: Cannot Undo Beyond This Point

What can I do for you?

```
Invalid command format!
help: Shows program usage instructions.
Example: help
```

*Figure 2. History Section of UI*

This command is to facilitate the undo/redo commands, as it becomes easier for you to track what changes were made before, and which commands can be undone/redone. This is especially useful when many commands have been executed and it is difficult to remember the sequence of execution of the commands.

Executing the `history` command will bring you to the "History" section of Alfred, which displays all the previously-executed commands as panels. There are 3 main delimiters. Each delimiter is just a visual representation of the limits for the undo/redo commands.

You cannot redo any command beyond the "redo" delimiter, or undo any command beyond the "undo" delimiter. The "current" delimiter tells you where you are at relative to the rest of the commands you have executed. It represents the current state of the data.

Note that only commands that change the state of the data in Alfred will be displayed in the "History" section and are undo/redo-able. For instance, `list participants` will not be undo/redo-able, as it simply shows you the participants in Alfred and does not change any information in Alfred. On the other hand, invoking `add participant` with the suitable parameters will be undo/redo-able and will be shown in the "History" section by the `history` command.

In total, only 50 states will be stored, so this serves as a limit for the number of commands you can undo/redo to.

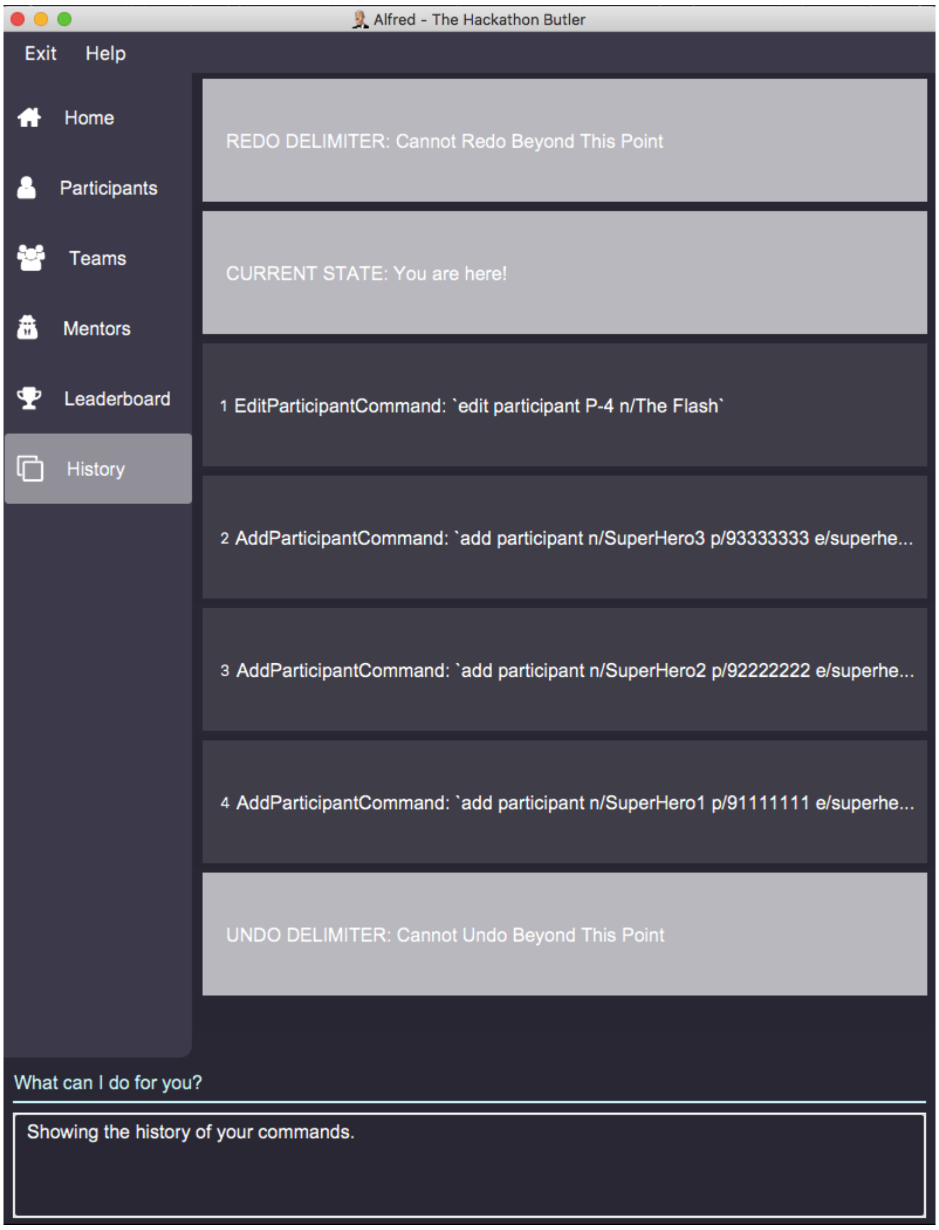
**NOTE** The following commands are not undo/redo-able: `help`, `list`, `find`, `history`, `leaderboard`, `getTop`, `export`, `help`, `home`, `undo`, `redo`. All other commands are undo/redo-able.

Example:

After running the following commands:

1. `list participants`
2. `add participant n/SuperHero1 p/+6591111111 e/superhero1@gmail.com`
3. `add participant n/SuperHero2 p/+6592222222 e/superhero2@gmail.com`
4. `add participant n/SuperHero3 p/+6593333333 e/superhero3@gmail.com`
5. `edit participant P-4 n/The Flash`

Running `history` will show you the following output in the "History" section of the Graphical User Interface:



REDO DELIMITER: Cannot Redo Beyond This Point

CURRENT STATE: You are here!

1 EditParticipantCommand: `edit participant P-4 n/The Flash`

2 AddParticipantCommand: `add participant n/SuperHero3 p/93333333 e/superhe...

3 AddParticipantCommand: `add participant n/SuperHero2 p/92222222 e/superhe...

4 AddParticipantCommand: `add participant n/SuperHero1 p/91111111 e/superhe...

UNDO DELIMITER: Cannot Undo Beyond This Point

What can I do for you?

Showing the history of your commands.

*Figure 3. Example Output from History Command*

The topmost panel is the "redo" delimiter. The second panel from the top is the "current" delimiter. The bottommost panel shows you the "undo" delimiter.

In this example, the output of the `history` command shows you can invoke the `undo` command four times.

Notice that the `list participants` command is not shown in the "History" section as it does not change data.

Also note that the undo-able commands are numbered, with the 1st undo-able command being the most recently executed command (the EditParticipantCommand), and the 4th undo-able command being the oldest executed command. In this case, the maximum number of commands you can undo at once is 4.

Also note that in this case, no commands are redo-able, that’s why there are no panels between the "redo" and "current" delimiters. Hence, executing `redo` command will result in an error.

# 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. The following text is taken from the Undo/Redo feature section of the Developer Guide, and explains how the Undo/Redo feature works. For more information on the rest of the features I implemented, please feel free to look through the developer guide here.*

## 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.

### 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.

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

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

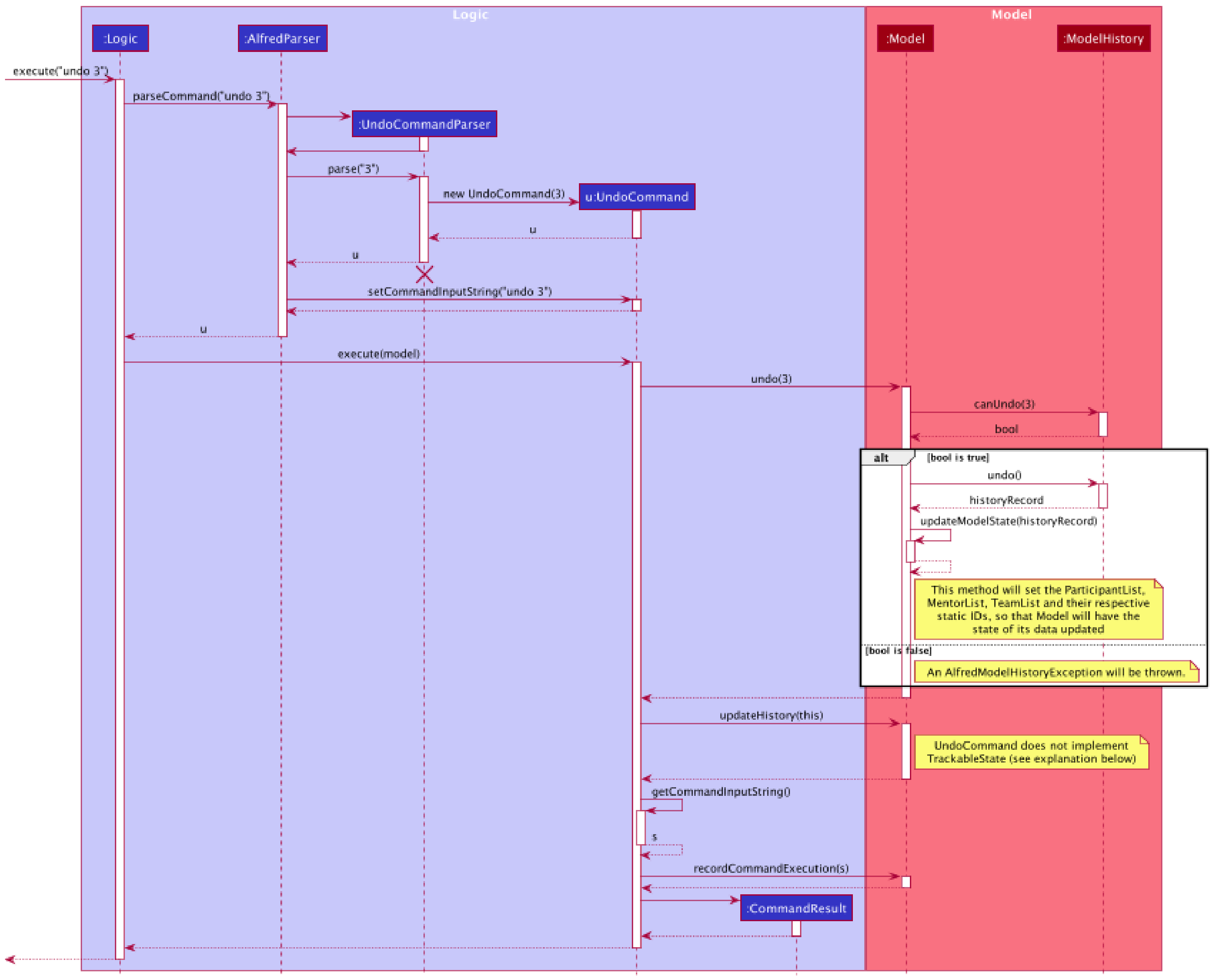

Logic
Model
:Logic
:AlfredParser
:Model
:ModelHistory
execute("undo 3")
parseCommand("undo 3")
:UndoCommandParser
parse("3")
new UndoCommand(3)
u:UndoCommand
u
u
setCommandInputString("undo 3")
u
execute(model)
undo(3)
canUndo(3)
bool
alt
[bool is true]
undo()
historyRecord
updateModelState(historyRecord)
This method will set the ParticipantList, MentorList, TeamList and their respective static IDs, so that Model will have the state of its data updated
[bool is false]
An AlfredModelHistoryException will be thrown.
updateHistory(this)
UndoCommand does not implement TrackableState (see explanation below)
getCommandInputString()
s
recordCommandExecution(s)
:CommandResult

*Figure 4. 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.

## 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.**

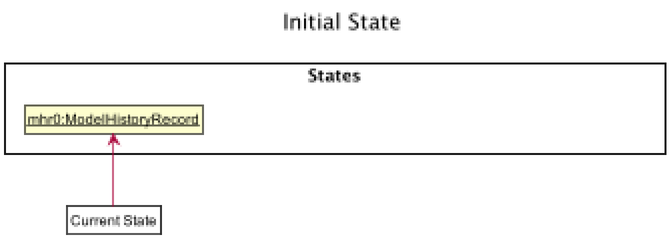


*Figure 5. Initial State of* `ModelHistoryManager`

**This is what happens after each step:**

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

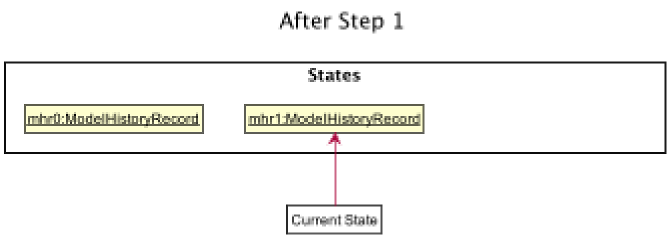


*Figure 6. State of* `ModelHistoryManager` *after Step 1*

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`

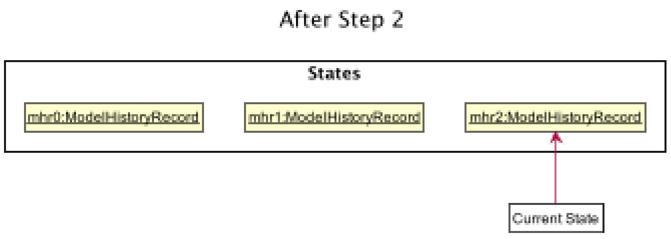


*Figure 7. State of* `ModelHistoryManager` *after Step 2*

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`

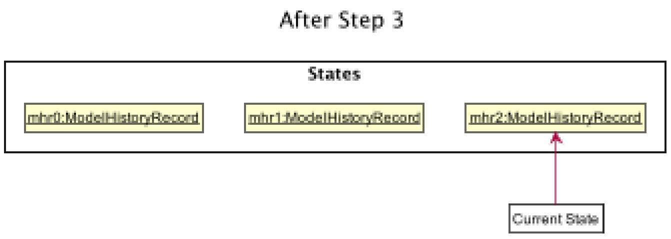


*Figure 8. State of* `ModelHistoryManager` *after Step 3*

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`

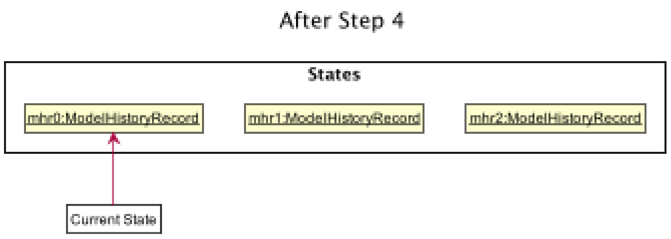


*Figure 9. State of* `ModelHistoryManager` *after Step 4*

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`

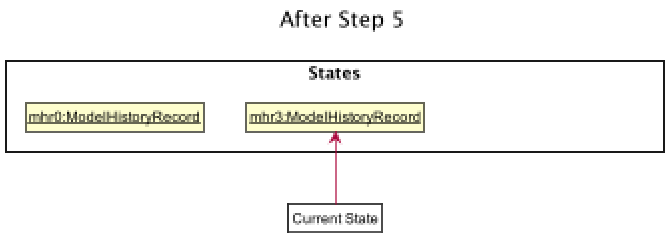


*Figure 10. State of* `ModelHistoryManager` *after Step 5*

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.

## 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.