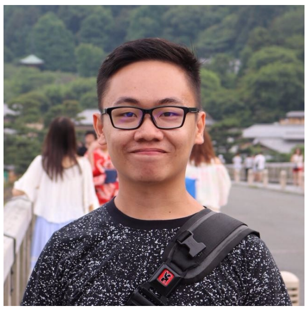
Keith Fong - Project Portfolio



Github | LinkedIn | Website

- Role - Team Leader
- Responsibilities
    - UI
    - Scheduling & tracking
    - Threading

PROJECT: Car park Finder

Overview

Car park finder is a desktop address book application to find HDB (Housing Development Board) car parks in Singapore. It allows you to work with a **Command Line Interface (CLI)** to display a list of car parks with the convenience of simply typing. If you **use the computer frequently** and **commute by driving**, you would find our application useful in helping you obtain various information about car parks.

Summary of contributions

- **Major enhancement**: Modified the **find feature to search for partial words** and **ignore certain words**.
    - What it does: Allows the user to search for partial instances of words without needed to type everything.
    - Justification: This feature improves the product significantly because a user can type lesser and the app should provide the same data still.
    - Highlights: This enhancement affects the existing commands and commands to be added in the future. I also wrote extra code which allowed us to re-use in the future.
- **Minor enhancements**:
    - Pulled postal code information using coordinates data by querying an API to convert it to store in a file for quick loading.

    - Updated HTML,CSS and Javascript for google maps display with real time updates from each individual commands.
    - Credits: [SVY21 to WGS84].

- **Code contributed**: [String Util], [Find Command], [Find Command Parser], [Gson test], [HTML, JS].
- **Other contributions**:
    - Project management:
        - Morphed the code from address book to car park. #1, #5
        - Morphed the test cases from address book to car park. #15, #25
    - Enhancement of existing features:
        - Wrote additional test for existing features to increase coverage from 72% - 77% #83
    - Documentation:
        - Did cosmetics tweaks to existing contents of the User Guide: (Pull requests: #34, #36, #60, #74, #76)
    - Community:
        - PRs reviewed (with non-trivial review comments): #28, #37

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.

Finding car parks : `find`

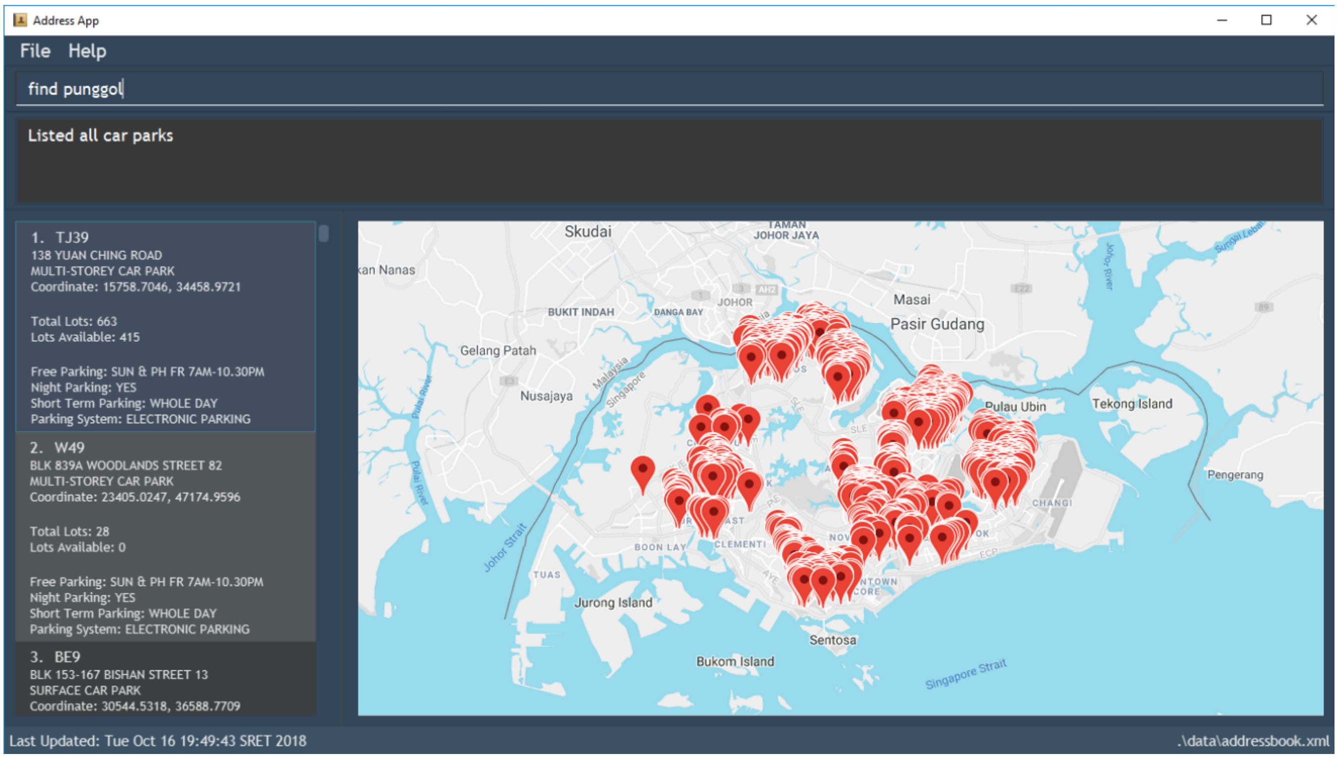
Find a list of car parks within a certain location.

Format	Abbreviation	Example(s)

`find KEYWORD`	`f`	`find serangoon`
		`f HG83`

NOTE

- Common words are ignored, like blk and ave.
- Upper and lower case characters do not matter.


Address App

File Help

find punggol

Listed all car parks

1. TJ39
138 YUAN CHING ROAD
MULTI-STOREY CAR PARK
Coordinate: 15758.7046, 34458.9721

Total Lots: 663
Lots Available: 415

Free Parking: SUN & PH FR 7AM-10.30PM
Night Parking: YES
Short Term Parking: WHOLE DAY
Parking System: ELECTRONIC PARKING

2. W49
BLK 839A WOODLANDS STREET 82
MULTI-STOREY CAR PARK
Coordinate: 23405.0247, 47174.9596

Total Lots: 28
Lots Available: 0

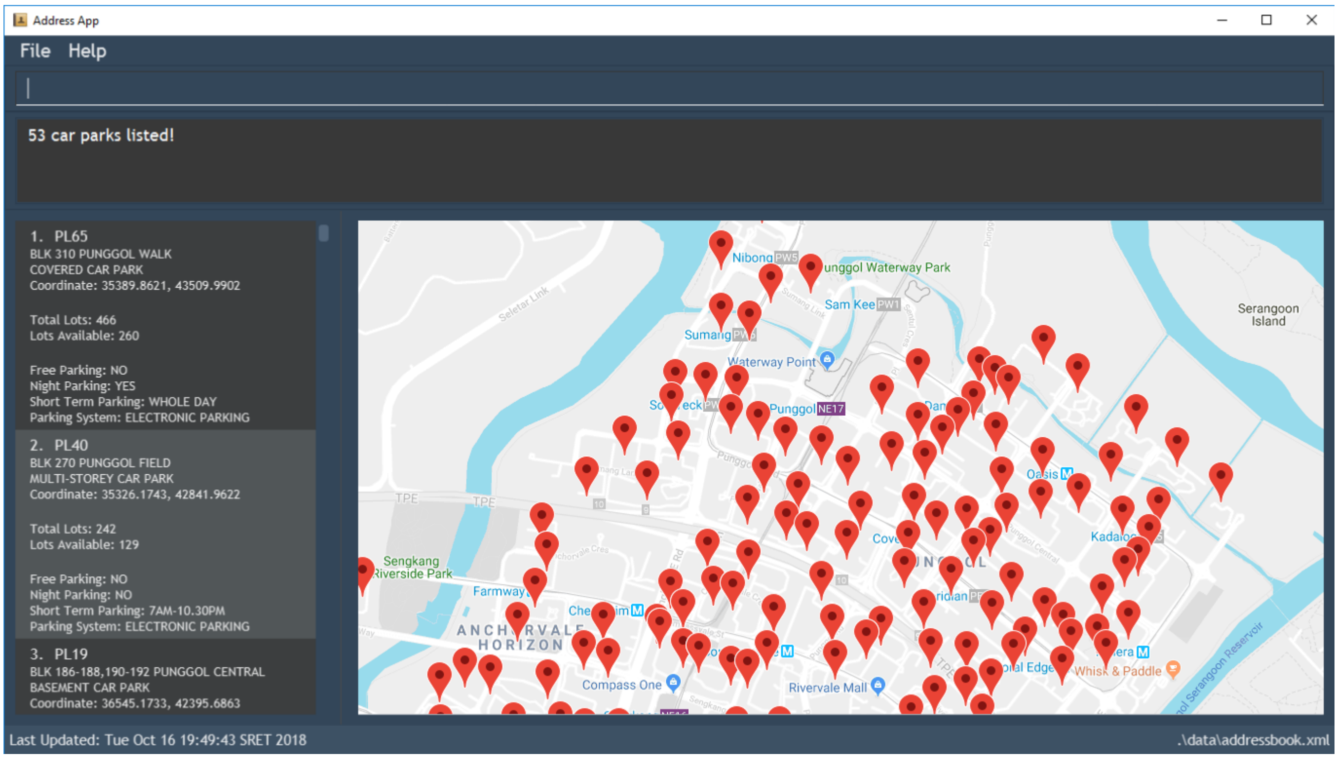
Free Parking: SUN & PH FR 7AM-10.30PM
Night Parking: YES
Short Term Parking: WHOLE DAY
Parking System: ELECTRONIC PARKING

3. BE9
BLK 153-167 BISHAN STREET 13
SURFACE CAR PARK
Coordinate: 30544.5318, 36588.7709

Last Updated: Tue Oct 16 19:49:43 SRET 2018 .\data\addressbook.xml

Figure 1. Before using find command

The figure above is a sample display of the application.


Address App

File Help

53 car parks listed!

1. PL65
BLK 310 PUNGGOL WALK
COVERED CAR PARK
Coordinate: 35389.8621, 43509.9902

Total Lots: 466
Lots Available: 260

Free Parking: NO
Night Parking: YES
Short Term Parking: WHOLE DAY
Parking System: ELECTRONIC PARKING

2. PL40
BLK 270 PUNGGOL FIELD
MULTI-STOREY CAR PARK
Coordinate: 35326.1743, 42841.9622

Total Lots: 242
Lots Available: 129

Free Parking: NO
Night Parking: NO
Short Term Parking: 7AM-10.30PM
Parking System: ELECTRONIC PARKING

3. PL19
BLK 186-188,190-192 PUNGGOL CENTRAL
BASEMENT CAR PARK
Coordinate: 36545.1733, 42395.6863

Last Updated: Tue Oct 16 19:49:43 SRET 2018 .\data\addressbook.xml

Figure 2. After using find command

The figure above shows what happen after executing the `find` command.

Undoing commands: `undo`

Restores the car park finder to the state before the previous undoable command was executed.

Format	Abbreviation	Example(s)

`undo`	`-`	`find sengkang`,`list`,`undo`
		`find sengkang`,`filter n\`,`undo`,`undo`

NOTE

- You can only `undo` when you have ran a commands.
- Undoable commands: those commands that modify the car park finder's content (`clear`, `find`, `filter` and `query`).

Redoing commands : `redo`

Reverses the most recent `undo` command.

Format	Abbreviation	Example(s)
`redo`	`-`	`find sengkang`,`undo`,`redo`

NOTE

- You can only `redo` after an `undo` command.

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.

Find feature

The find feature searches for a car park by keyword or location.

Overview

The find mechanism is facilitated by `FindCommand` and `FindCommandParser`. It extends `Command` and implements the following operations:

- `FindCommand#execute()` — Executes the command by running a predicate `CarparkContainsKeywordsPredicate` to update the car park list.

The find mechanism is supported by `FindCommandParser`. It implements `Parser` and contains the following operations:

- `FindCommandParser#parse()` — Checks the arguments for empty strings and throws a `ParseException` if empty string is found. It then splits it by one or more white spaces. It then removes any strings in the list of common words.

The predicate `CarparkContainsKeywordsPredicate` takes in a list of strings and checks if any of the strings matches the name or address of a car park fully or partially.

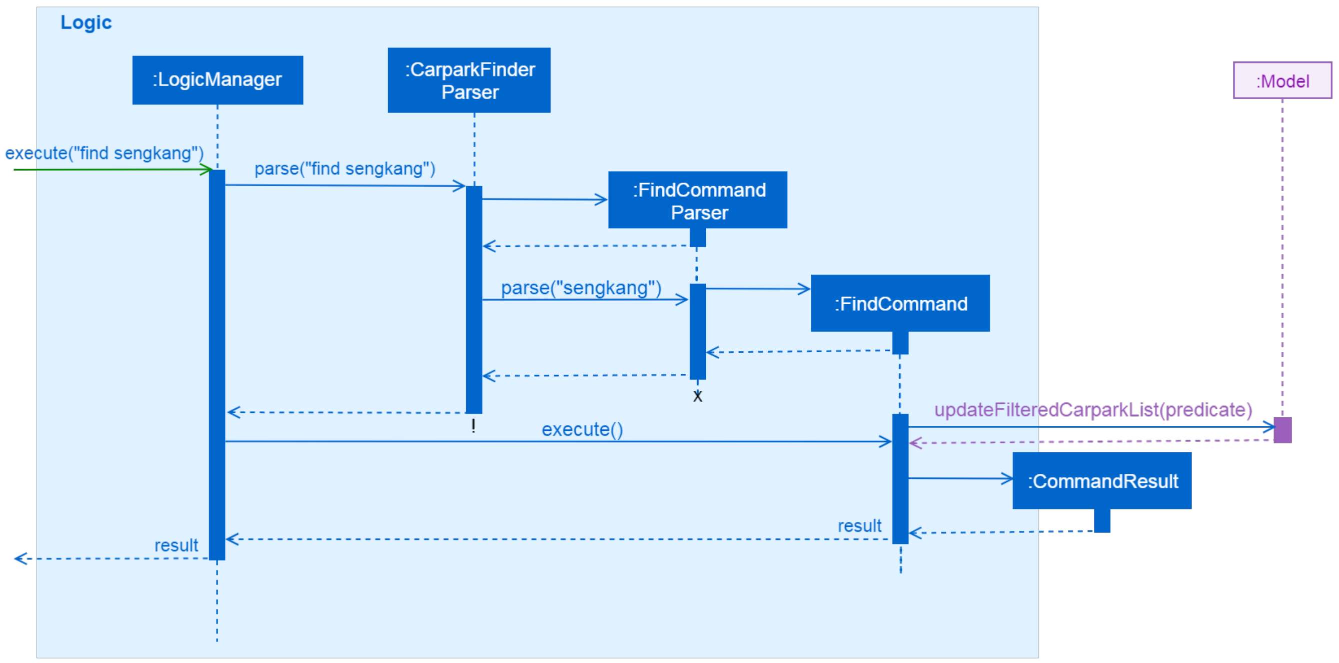


Figure 3. How the find operation works

The diagram above describes how the flow of a find command would execute. It rely on `FindCommandParser` to ensure the variables are correct.

Example

Given below is an example usage scenario of how the Find mechanism behaves at each step.

Step 1. The user launches the application for the first time.

Step 2. The user executes `find punggol` command to get all car parks in punggol. The `find` command calls `FindCommandParser#parse()`.

NOTE	If a command execution fails, it will not call `FindCommand#execute()`, and the car park finder state will not be saved.

Step 3. The entire list of car parks is filtered by the predicate `CarparkContainsKeywordsPredicate`, which checks for the keyword `punggol`.

Step 4. The filtered list of car parks is returned to the GUI.

The flow chart below describes the user interaction with the application and how it processes it.

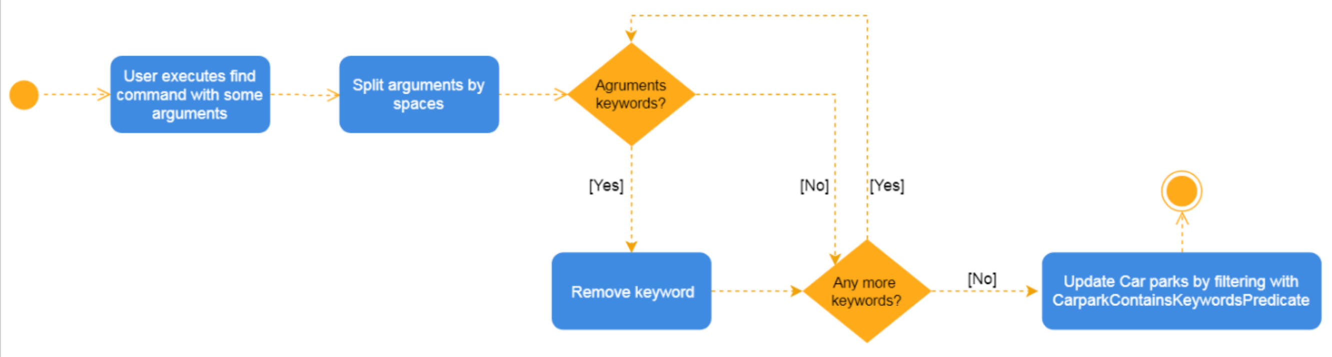

User executes find command with some arguments
Split arguments by spaces
Agruments keywords?
[Yes]
[No]
[Yes]
Remove keyword
Any more keywords?
[No]
Update Car parks by filtering with CarparkContainsKeywordsPredicate

Figure 4. Flow chart of the find operation.

Design Considerations

Aspect: How predicate works

- **Alternative 1 (Current choice):** Predicate have additional filter with an ignore list.

Pros	Re-usable functions introduced for partial checking.
Cons	Reading the car park list while querying might cause unintended side effects if not handled properly.

- **Alternative 2:** Filter the data when it is taken in.

Pros	Easy to maintain as predicate will have lesser conditions.
Cons	Breaks OOP style as the parser will modify the data.

[Proposed] Data Encryption

The car park data and user data will be encrypted to prevent users from editing and manipulating them.

Overview

The data encryption mechanism works by encrypting the information by a unique key generated by every users individual system. The key will stored in a secured location to prevent people from accessing it.

The two main files it will encrypt are:

- Carpark information
- User’s favorites

Example

This feature is coming in v2.0.

Undo/Redo feature

Current Implementation

The undo/redo mechanism is facilitated by `VersionedCarparkFinder`. It extends `CarparkFinder` with an undo/redo history, stored internally as an `carparkFinderStateList` and `currentStatePointer`. Additionally, it implements the following operations:

- `VersionedCarparkFinder#commit()` — Saves the current car park finder state in its history.
- `VersionedCarparkFinder#undo()` — Restores the previous car park finder state from its history.
- `VersionedCarparkFinder#redo()` — Restores a previously undone car park finder state from its history.

These operations are exposed in the `Model` interface as `Model#commitCarparkFinder()`, `Model#undoCarparkFinder()` and `Model#redoCarparkFinder()` respectively.

Given below is an example usage scenario and how the undo/redo mechanism behaves at each step.

Step 1. The user launches the application for the first time. The `VersionedCarparkFinder` will be initialized with the initial car park finder state, and the `currentStatePointer` pointing to that single car park finder state.

The following diagram showcases the state at the start of the program.

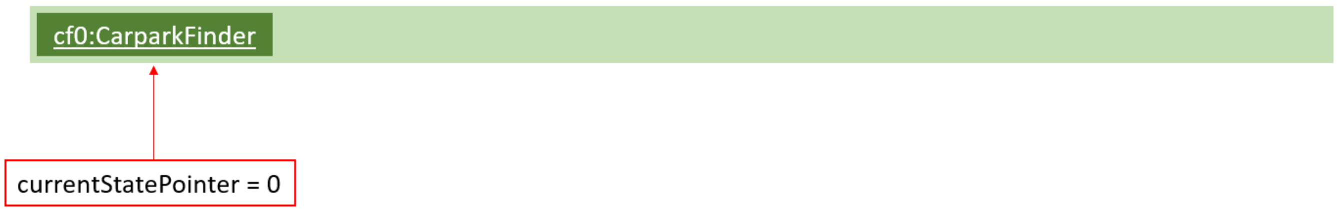


Figure 5. State and the start of the program

Step 2. The user executes `find sengkang` command to find list of car park which contain sengkang from the car park finder. The `find` command calls `Model#updateFilteredCarparkList()`, causing the modified state of the car park finder after the `find sengkang` command executes to be saved in the `carparkFinderStateList`, and the `currentStatePointer` is shifted to the newly inserted car park finder state.

The following diagram shows a new state is created after the command `find segkang` is ran.

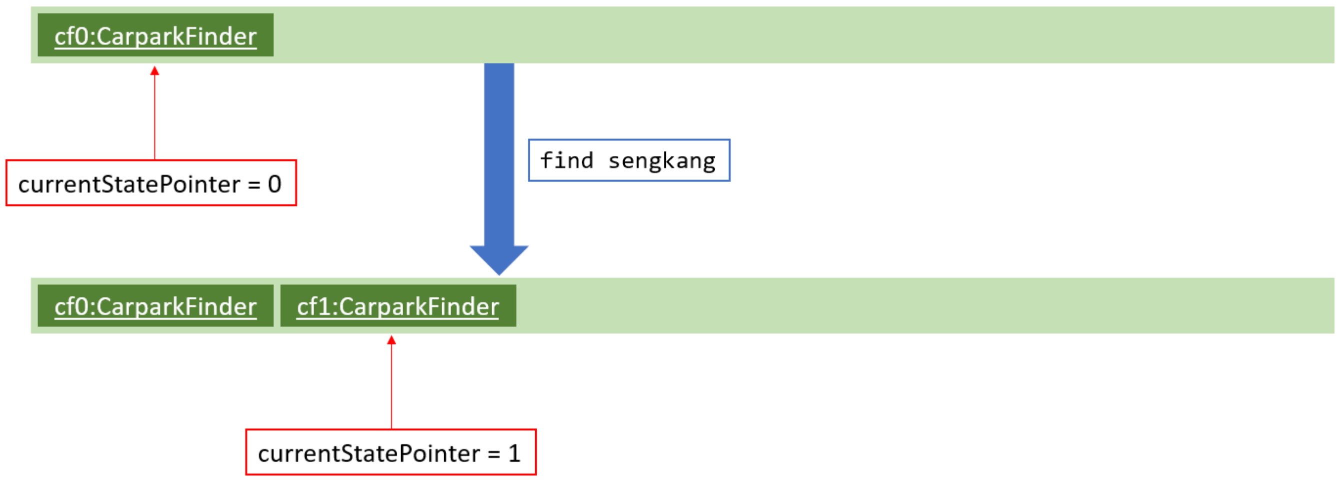


Figure 6. State after running find command

Step 3. The user executes `clear` to clear all entries. The `clear` command also calls `Model#commitCarparkFinder()`, causing another modified car park finder state to be saved into the `carparkFinderStateList`.

The following diagram shows a new state is created after the command `clear` is ran.

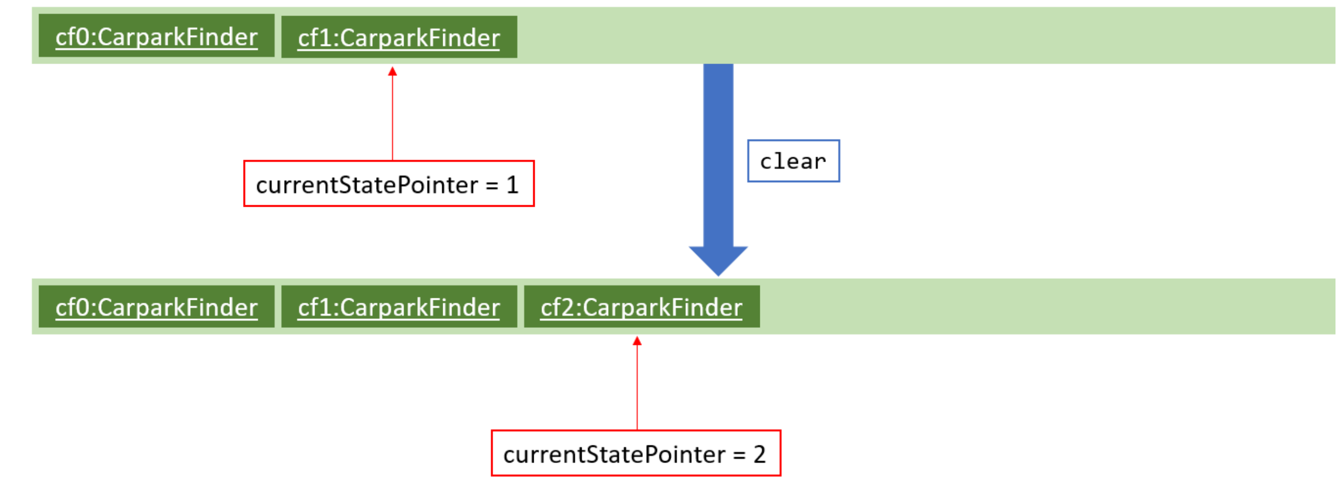


Figure 7. State after running find then clear command

NOTE	If a command fails its execution, it will not call `Model#commitCarparkFinder()`, so the car park finder state will not be saved into the `carparkFinderStateList`.

Step 4. The user now decides that adding the person was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoCarparkFinder()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous car park finder state, and restores the car park finder to that state.

The following diagram shows a new state is created after the command `undo` is ran. The state pointer is moved.

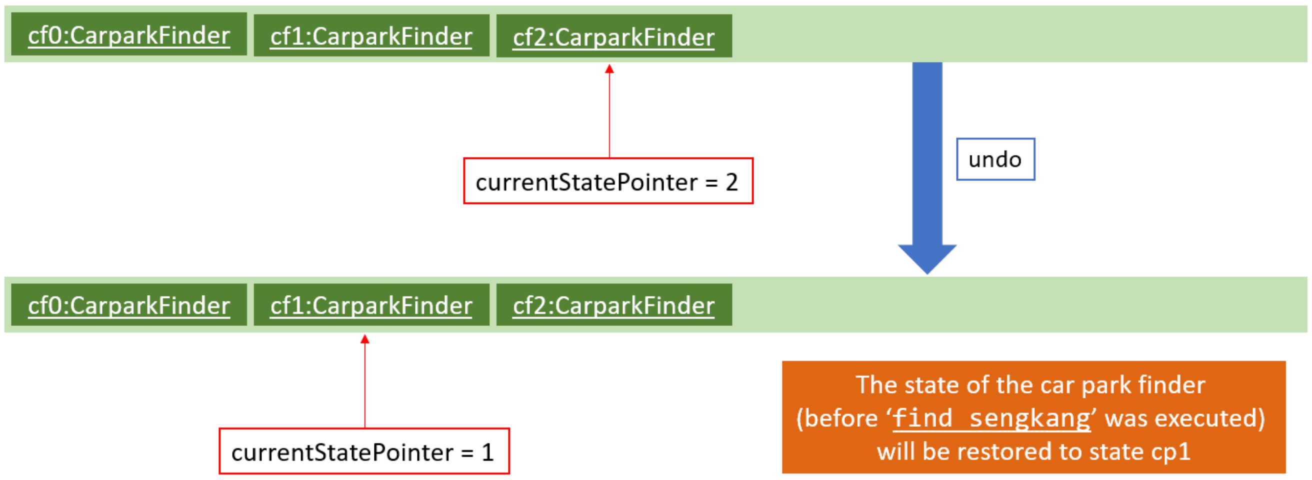


Figure 8. How the undo feature works in a diagram

NOTE	If the `currentStatePointer` is at index 0, pointing to the initial car park finder state, then there are no previous car park finder states to restore. The `undo` command uses `Model#canUndoCarparkFinder()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

The following ***sequence diagram*** shows how the undo operation works:

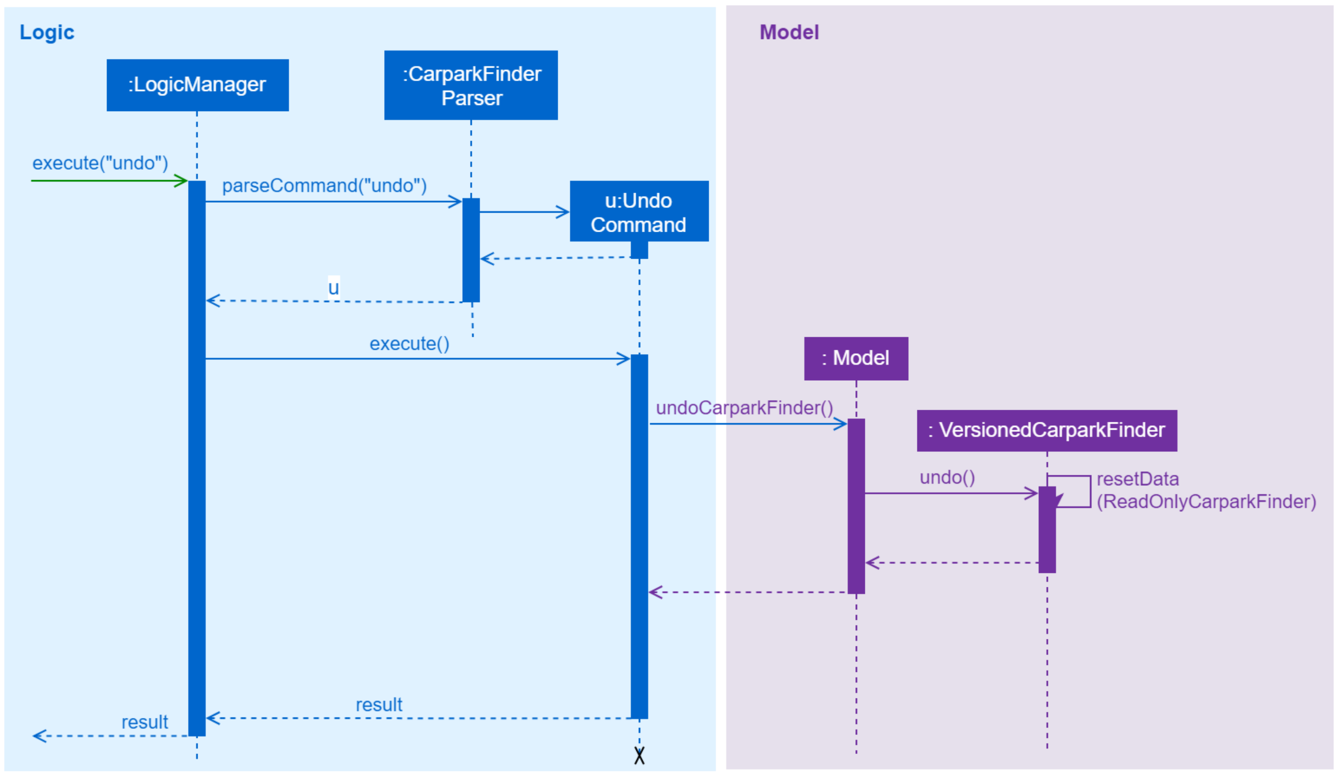


Figure 9. Sequence diagram of Undo/Redo feature

The `redo` command does the opposite — it calls `Model#redoCarparkFinder()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores the car park finder to that state.

NOTE If the `currentStatePointer` is at index `carparkFinderStateList.size() - 1`, pointing to the latest car park finder state, then there are no undone car park finder states to restore. The `redo` command uses `Model#canRedoCarparkFinder()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the redo.

Step 5. The user then decides to execute the command `list`. Commands that do not modify the car park finder, such as `list`, will usually not call `Model#commitCarparkFinder()`, `Model#undoCarparkFinder()` or `Model#redoCarparkFinder()`. Thus, the `carparkFinderStateList` remains unchanged.

The follow diagram showcases what happen when a command that does not modify the state is used.

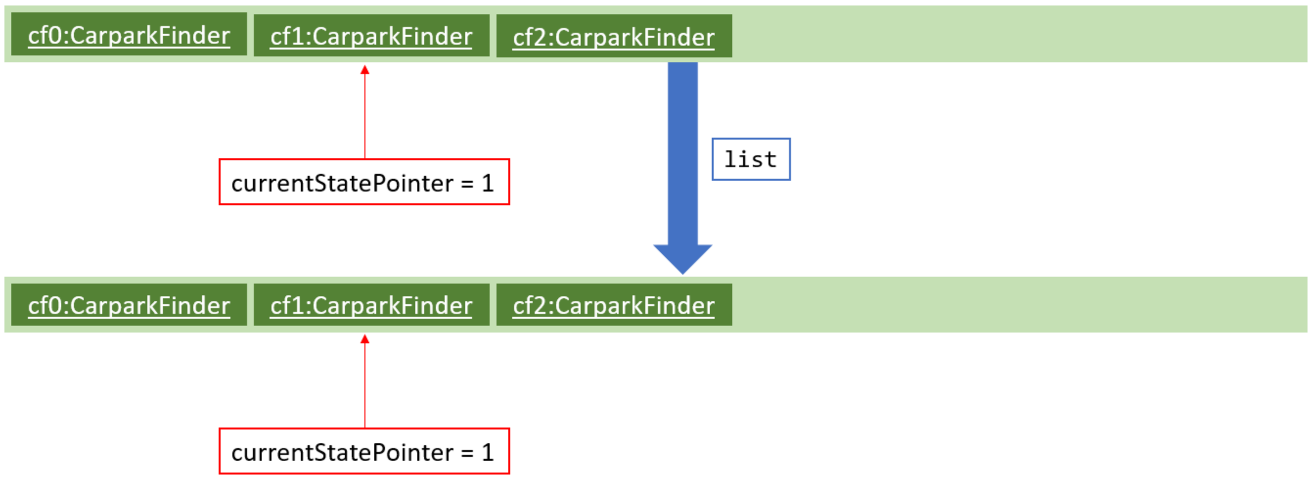


Figure 10. As the command does nothing, the state is not changed, but a new state is introduced

Step 6. The user executes `clear`, which calls `Model#commitCarparkFinder()`. Since the `currentStatePointer` is not pointing at the end of the `carparkFinderStateList`, all car park finder states after the `currentStatePointer` will be purged. We designed it this way because it no longer makes sense to redo the `find sengkang` command. This is the behavior that most modern desktop applications follow.

The following diagram showcases when a new command is used after an undo.

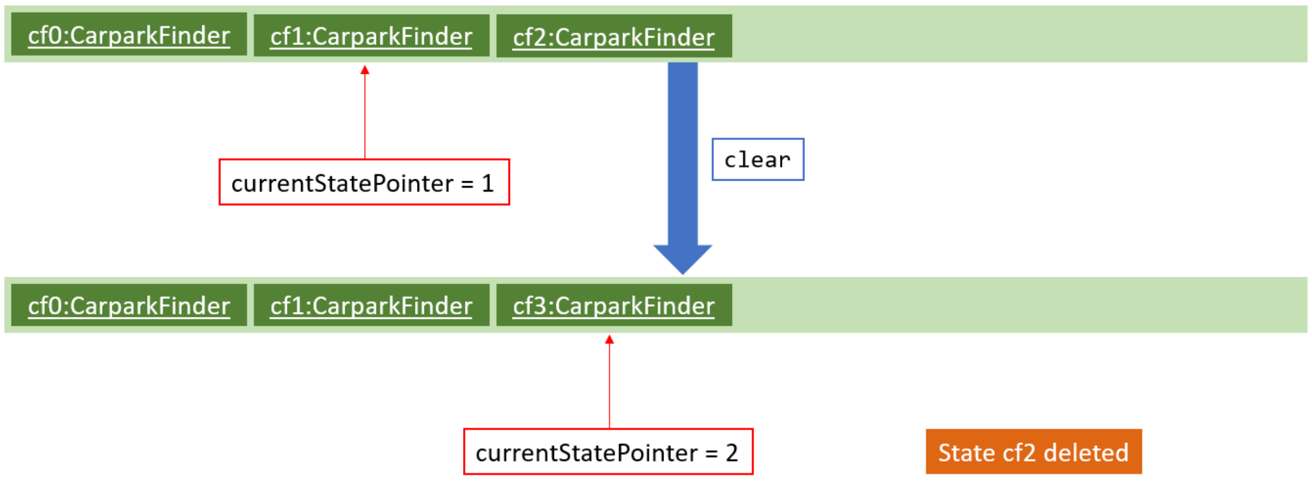


Figure 11. Deleting a state diagram after an undo

The following activity diagram summarizes what happens when a user executes a new command:

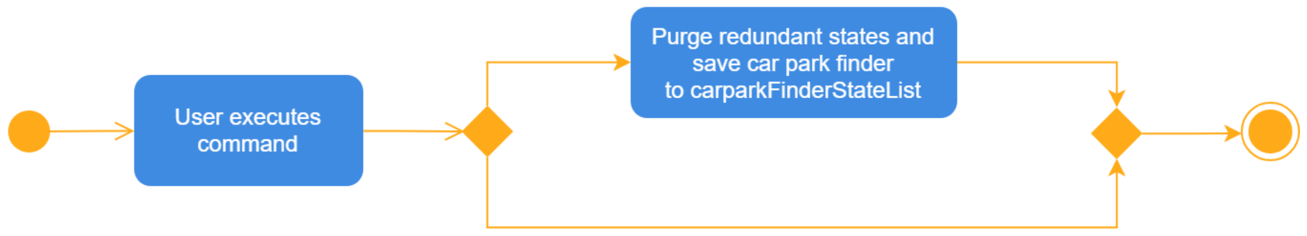

User executes command
Purge redundant states and save car park finder to carparkFinderStateList

Figure 12. Activity diagram of a sample command

Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire car park finder.

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 `select`, just save the car park being selected).
Cons	We must ensure that the implementation of each individual command are correct.

Aspect: Data structure to support the undo/redo commands

- **Alternative 1 (current choice):** Use a list to store the history of car park finder states.

Pros	Easy for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project.
Cons	Logic is duplicated twice. For example, when a new command is executed, we must remember to update both `HistoryManager` and `VersionedCarparkFinder`.

- **Alternative 2:** Use `HistoryManager` for undo/redo

Pros	We do not need to maintain a separate list, and just reuse what is already in the codebase.
Cons	Requires dealing with commands that have already been undone: We must remember to skip these commands. Violates Single Responsibility Principle and Separation of Concerns as `HistoryManager` now needs to do two different things.