Joshua Wong - Project Portfolio

PROJECT: Alfred - The Hackathon Butler

Overview

As part of our software engineering project, my team of five, including me, were tasked with either improving the existing **AddressBook3** application codebase, or morphing it into any application, as long as it maintained a command line interface and had a Graphical Display. We opted for the latter and morphed the **AddressBook3** codebase into a Hackathon manager, named **Alfred** that is targeted at Hackathon HR managers to aid them in the organization of hackathons, including the logistics and administration. **Alfred** packages multiple features into a single application, such as but not including judging, data validation, find, undo/redo, a whole new UI and more!

For **Alfred**, I was mainly in charge of the Model backend, the find command and the enhancement of existing commands.

Summary of contributions

- **Major Enhancement**: Added **the command to perform advanced finding functions on the stored data**
    - What it does: This functionality enables the user to search for users in the database via different fields and complements the preexisting `view` command, which only gets a single user by `ID`.
    - Justification: This functionality makes searching for users extremely convenient, especially if the hackathon has hundreds of participants. It also can return multiple users that match the given search parameters in a single command, which makes it more efficient than the more specialized `view` command and removes the need for the hackathon organizer to remember randomly generated `ID` numbers. This functionality makes **Alfred** a viable alternative to existing software like Excel.
    - Highlights: Like existing query langages, the find command supports a powerful search, allowing users to search by intersection, union across multiple fields and to perform a negative search on selected fields.
- **Major Enhancement**: Refactored and Implemented the new `Model` for **Alfred**
    - What it does: The preexisting **AddressBook3** codebase only has the relevant objects for the address book application. However, **Alfred** required new objects such as `Score`, `Participant` and `Team`, and to create new methods for the other parts of the codebase to interact with this in-memory data. It is also the only component of the codebase that interacts with `Storage`.
    - Justification: Without the new objects created by `Model`, it would also impossible to store anything in memory. The methods exposed by `Model` are the only way that the other parts of the object can interact with the in-memory data. Thus, `Model` serves as a single source of

truth for the application in order to ensure that the hackathon information managed by the application is accurate.

- **Minor enhancement**: Updated the help command UI to reflect the new commands in **Alfred**
- **Code contributed**:
  - Functional code
  - Test code
- **Other contributions**:
  - Project management:
    - Managed release `v1.1`
    - Set up the repository and continous deployment of documentation
    - Deprecated old **AddressBook3** code (#160, #169, #316)
  - Tests:
    - Wrote tests that helped to raise the coverage of the codebase by 8% (#187, #316)
  - Documentation:
    - Updated User Guide and Developer Guides to reflect features implemented (#145, #200)
  - Community:
    - Maintained the develop branch on Github in the initial stages when the master branch was broken
    - Reviewed PRs with constructive comments
    - Participated in discussions with teammates , especially about the overall architecture and plan for the application
    - Participated in dogfooding exercises and reported any bugs found to teammates
  - Tools:
    - In my capacity as Github In-Charge, helped to integrate the following tools
    - Integrated codecov into the repository to enable automated test coverage detection
    - Integrated Travis CI into the repository to enable continuous integration
    - Integrated Mockito testing library for easier automated testing of the codebase

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 sections I added mainly pertained to the guide for the find feature.

Finding a Specific Entity: `find {mentor/team/participant} n/NAME otherParameters`

Searches for Entities by selected fields, instead of their ID, in case you find that the ID is difficult to keep track of.

> - Take note that the `find` command only searches and matches the fields whose strings are a substring of the given value. It also does an intersection search in the event where one or more fields are provided, that is, the entities found would have the selected values for each of the fields you key in. This search is case insensitive.

Examples:

- `find mentor n/Joshua Wong` will display a list of all mentors in the Hackathon who are named "Joshua Wong", or have "Joshua Wong" in their name.
- `find participant n/John Doe` will display a list of all participants in the Hackathon who are named "John Doe", or have "John Doe" in their name.
- `find team n/FutureHackathonWinner` will display a list of all teams in the Hackathon that are named "FutureHackathonWinner", or have "FutureHackathonWinner" in their name.

Each entity will have different fields available to find.

For participant, `n/NAME e/EMAIL p/PHONE` are all options to search for.

For team, `n/TEAMNAME pn/PROJECTNAME` are also all options to search for.

For mentor, `n/NAME, e/EMAIL, p/PHONE o/ORGANIZATION` are all available

Example for multiple fields:

- `find participant n/Damith e/damith.com` finds all participants whose name contains the string "Damith" and whose emails contain "damith.com"

Advanced Finding of a Specific Entity by Union or Negative: `find [AND/OR] {mentor/team/participant} n/NAME otherParameters [EXCLUDE] otherParameters`

The default find command for single and multiple fields works via a find by intersection. That is, entities must be true for all the predicates for it to be displayed. However, Alfred also supports finding by union.

As above, all find commands are case insensitive.

The commands for this is as such:

- `find participant OR n/Damith e/nus` will do a search for all Participants whose name contains "Damith" or whose email contains "nus" in it.

Also do note that for this command, the OR key must be placed before all the arguments to the command. Also, the OR key can be replaced by the AND key to do a search by intersection. If none are provided, then a search by intersection is done by default. The AND/OR keyword must be in caps.

Next, Alfred also supports negative searches, if you wish to do it. Simply run

- `find mentor n/Boss EXCLUDE e/boss.com` will return all mentors whose name has a "boss" and whose email does not contain "boss.com"

Also, you can do negative searches by union as well.

- `find mentor OR n/boss EXCLUDE e/boss.com` will now return all mentors whose name has a "boss" in it or whose email does not contain "boss.com"

However, there are also some caveats when it comes to using `find`.

1. The `AND/OR` keyword must be placed at the front before all parameters and the `EXCLUDE` keyword
2. Anything after the `EXCLUDE` keyword will be processed using negative find.
3. No `AND/OR` keywords are allowed after the `EXCLUDE` keyword.
4. You can only search by `AND` or `OR`. You cannot do a search by both `AND` and `OR`

> Some notes on the logic used in find.
>
> - The logic used in find obeys normal boolean, probabilistic logic.
> - Hence commands like `find participant n/Ki EXCLUDE n/Ki` will return an empty list
> - Likewise, commands like `find participant OR n/Ki EXCLUDE n/Ki` will return the full list.

Other examples of valid commands are also provided here for your reference:

1. `find team OR EXCLUDE n/ArsenalFC pn/Football` will do a search of all teams whose name does not contain "ArsenalFC" or whose project name does not contain "Football".
2. `find participant AND n/Abramov EXCLUDE e/react` will do a search where participant names contain "Abramov" and whose email does not contain "react". In this case, the `AND` keyword could have been omitted because the default find does a search by intersection.

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. Apart from the excerpts provided below, I also contributed to the Model Section of the developer guide.

Model component

High Level Design Overview

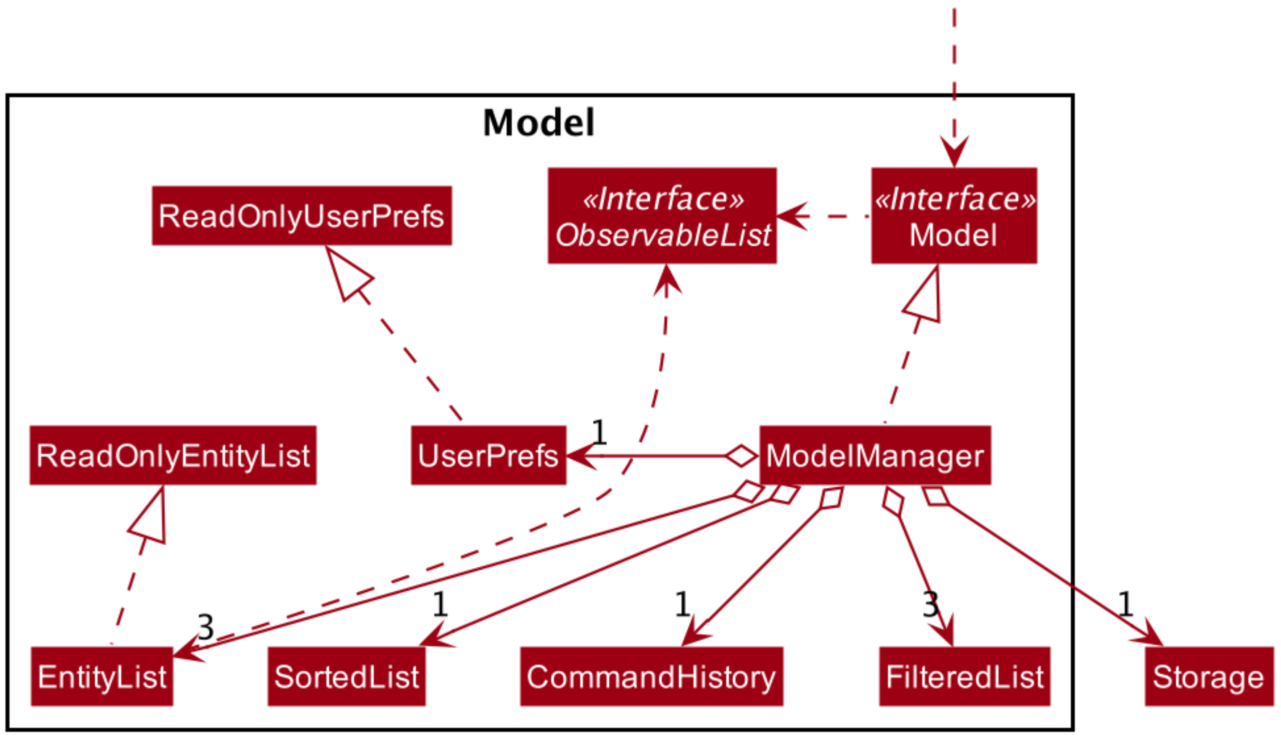


Figure 1. Structure of the Model Component

API : `Model.java`

The `Model`,

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

`ModelManager`

- ModelManager implements all the methods exposed by the Model Interface. The 3 most important aspects for its in-memory storage and UI are the `FilteredList`, `EntityList` and `UserPrefs` objects. As mentioned above, `ModelManager` also consists of other components, but these are not reflected in the diagrams for brevity and clarity.

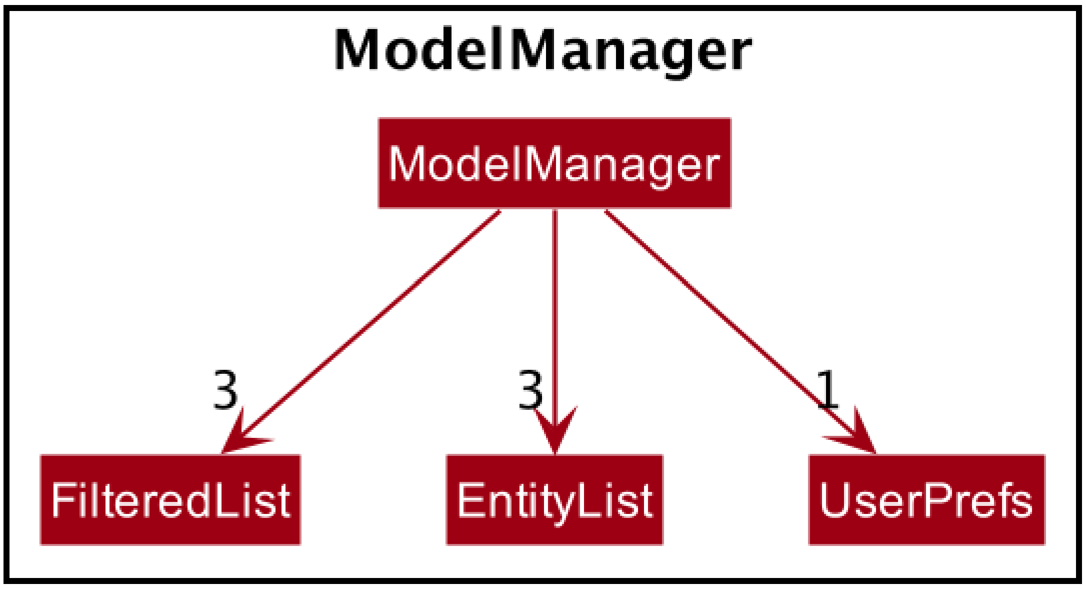


Figure 2. Simple Illustration of ModelManager

Each `EntityList` is also further subclassed into `ParticipantList`, `MentorList`, `TeamList`. Each of these lists can be seen as an individual address book from the original AB3 project. In turn, these `EntityList` objects contain the respective `Participant`, `Mentor` and `Team` objects. You can see the diagrams for the aforementioned 3 objects in our developer guide here.

Usage

When the ModelManager object is first created upon starting the application, the existing data is loaded from the disc via methods on the `Storage` object. However, if there are any bugs in the process, perhaps due corrupted data, a new `EntityList` is instantiated rather than run the risk of working with outdated data.

Due to its role as the API of the application, all calls which require access to the `Entity` objects will be done through `Model` and not via the lists directly. These operations are listed as public methods on the `Model` interface.

For operations which would entail mutating the data within the `EntityList` objects in any form, `Model` automatically communicates with the `Storage` object to save the data. The saving logic can be found within the `Storage` object and thus `Model` only needs to pass it `EntityList` objects on its end. The same applies for the other attributes in `Model`, such as `CommandHistory`; `Model` will automatically communicate with it for you.

If there are any errors along the way, it will be logged but the error would be handled within `Model` itself. Moreover, if there is an during a model operation, it will not be saved.

Design Considerations

1. Synchronization of data
    - The role of `ModelManager` is to ensure that the data is in sync with each other across all 3 `EntityList` objects. The reason behind this is because for example, the `Participant` object in

  `ParticipantList` is a separate object from the one inside `Team`. It was not possible due to make the `Participant` object hold a reference to team due to serialization issues on `Storage`.
  - As such, for each CRUD operation, `ModelManager` has to perform validation to ensure that the data modified/added is sync across all 3 `EntityList` objects.
  - This was also the reason why `Storage` was moved into the `Model` object, as in the current implementation of Alfred, only `ModelManager` needs to be aware of `Storage` to ensure that only it can save/read data. This would hence help to synchronize data better.

2. Single Responsibility Principle and Inheritance
   - Each class in `Model` is only responsible for a single task. For example, `TeamList` is only concerned with managing the `Team` entities stored in it. This would help to improve testability and code quality, especially since the size of the `Model` codebase is substantial
   - Inheritance was used to show links between related objects. For `Model`, the two related objects are `EntityList` and `Entity`. Inheritance was used to show this relationship and to reduce the need for code duplication.
3. Open Closed Principle
   - `Model` exposes many functions. However, in line with the Open Closed Principle, modifications to `Model` come in the form of exposing new methods on it and creating new attributes on the `ModelManager` object. The methods on `ModelManager` were also implemented as simply as possible so that future methods can build on them. This way, future modifications do not need to edit existing code, reducing the likelihood of regression bugs.
4. Design of the `Entity` objects
   - **Alternative 1**: Make the `Team` object the single source of truth (only `Team` has references to `Participant` and `Mentor`)
     - Pros: This would facilitate the serialization on `Storage`
     - Cons: As `Participant` and `Mentor` objects no longer hold a reference to the `Team` object, it is now possible for their fields to be different from their counterparts stored in `Team`, requiring Alfred to do significant validation
   - **Alternative 2**: Make `Participant`, `Mentor` and `Team` objects store a bidirectional reference to each other
     - Pros: The `Participant` objects in the `Team` 's participants field are exactly the same objects stored in the `ParticipantList`, reducing the need for validation code as they will never be out of sync
     - Cons: `Storage` serialization cannot handle bidirectional associations

We decided to opt for Alternative 1 as there was no easy solution to solve the issues `Storage` had with bidirectional associations. Also, the validation code for Alternative 1 was implemented early and employed many defensive programming practices, reducing the likelihood of bugs occurring due to similar objects having data that is out of sync.

Future Extensions

1. As the single source of truth for the application in runtime, there are many small functions on `ModelManager` now. These functions are implemented directly in the file itself. In the future, it

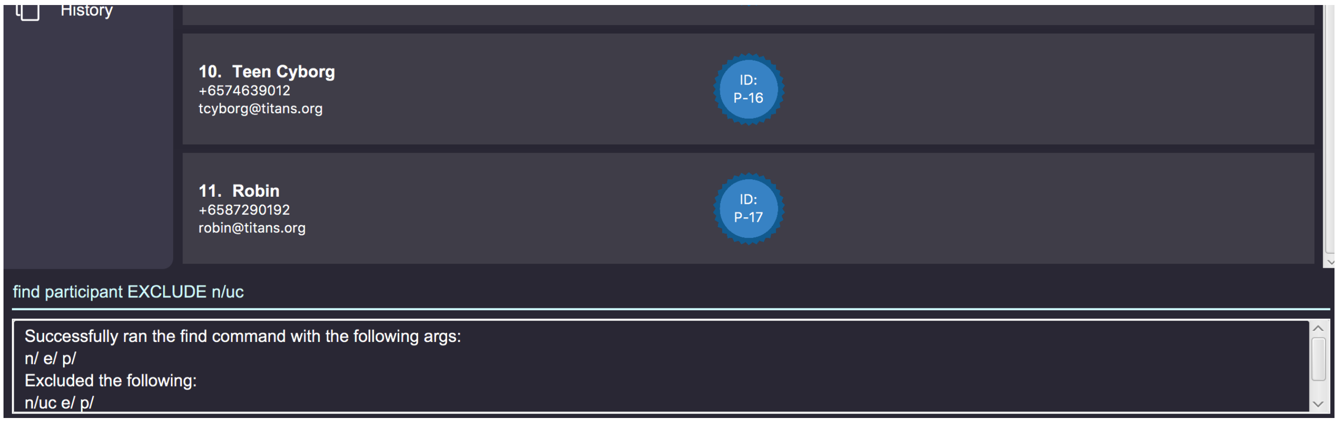
may be better to abstract these functions out into smaller modules as per the Dependency Inversion Principle. It was not done for v1 of Alfred as refactoring these methods would block developers and slow down feature development velocity. However, as Alfred scales, it is recommended that this refactoring be done.

Find Command

Currently, Alfred allows users to view specific entities by their `ID` using the `view ENTITY ID`. However, it may not be convenient for users to remember the `ID` of specific users, especially since `ID` objects are randomly generated.

To help Alfred become a viable alternative to Excel, Alfred also offers an improved `find` function that has been inspired by the power of Excel Macros. `find` offers a search by a single field and multiple fields. The list of fields that can be searched for for each entity can be found in the user guide, or by simply typing in an incorrect command in the application.

The find command also offers an option to do a negative search of the fields in the list. This can be done via the `EXCLUDE` keyword. All the parameters that come after the keyword will undergo a negative search. Figure 41 shows an example of this. `find participant EXCLUDE n/uc` will look for all participants whose name that does not contain "uc".



```
find participant EXCLUDE n/uc
```

```
Successfully ran the find command with the following args:
n/ e/ p/
Excluded the following:
n/uc e/ p/
```

Figure 3. Find Command User Interface

By default, Alfred does a search by intersection for the given entity. However, Alfred also allows the user to do a search by union. This can be done by inserting an `OR` keyword before all the parameters. Further examples on the usage of `find` can be found in the user guide.

Find Command Implementation Overview

The following sequence diagram shows the sequence of method calls used to display the filtered list on the application, from `Logic` to `ModelManager` upon the execution of a `FindParticipantCommand`. An analogous sequence diagram also applies for the `Team` and `Mentor` objects as well.

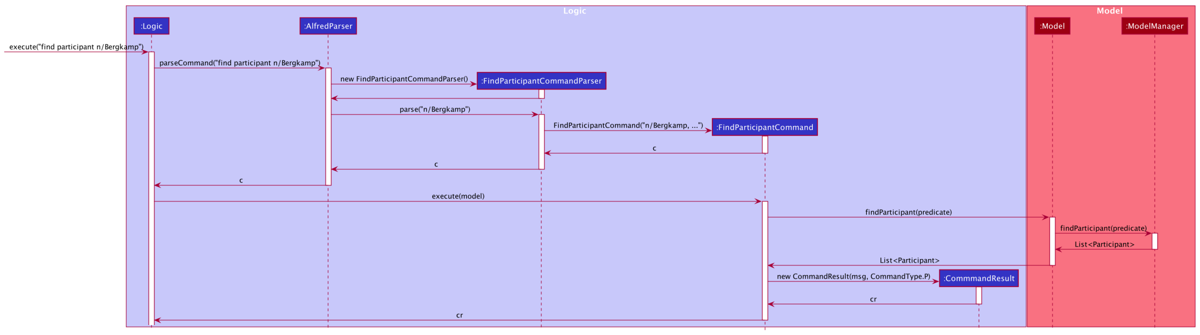

Logic
Model
:Logic
:AlfredParser
:Model
:ModelManager
execute("find participant n/Bergkamp")
parseCommand("find participant n/Bergkamp")
new FindParticipantCommandParser()
:FindParticipantCommandParser
parse("n/Bergkamp")
FindParticipantCommand("n/Bergkamp, ...")
:FindParticipantCommand
c
c
c
execute(model)
findParticipant(predicate)
findParticipant(predicate)
List<Participant>
List<Participant>
new CommandResult(msg, CommandType.P)
:CommmandResult
cr
cr

Figure 4. Find Participant Command Sequence Diagram

The left half of the diagram covers creation of the `FindParticipantCommand` while the right half details the interaction with the `Model`. There are 5 main steps in the execution of the command.

1. `FindCommandAllocator` allocates the correct parser for the find command based on the entity name provided. If the entity name is incorrect, then an error will be thrown and execution will terminate. This step is not represented in the sequence diagram to make it neater.
2. The `FindParticipantCommandParser` will then parse the String provided to it. If the string provided is invalid and does not follow the specified format, an error will be thrown. Likewise, if the find command is called and no parameters are provided, an error will also be thrown. During the parsing, the parser splits the input string given into two separate strings, one for normal parameters and the other for parameters that will undergo negated search. Two separate `ArgumentMultimap` objects are generated using the two strings. These `ArgumentMultimap` objects are the same as the ones found in AB3 and thus behave in a similar manner and have the same constraints.
3. If there are no errors, the `FindParticipantCommandParser` then creates a `FindParticipantCommand` which in turn generates the predicate it would use to filter the relevant `EntityList` by. These predicates are stored as static variables in a central `Predicates` file. `FindParticipantCommand` selects the predicates based on which fields are provided in the original input string.
4. With these predicates, it then calls the `findParticipant` method on the model with the generated predicate.
5. A list of `Participant` objects is then returned, and is also printed onto the console for the user's reference. Then, at `FindParticipantCommand`, a `CommandResult` is returned after the smooth execution of this `find` operation.

NOTE The find command does not mutate the list in any way. It only changes the `Entity` objects displayed using the generated predicate.

Design Considerations

The following are some design considerations for the Find Command Feature

Aspect: Format of the input string

- **Alternative 1**: Use a SQL style query string
  - Pros: SQL syntax is universally used to communicate with databases and is clean
  - Cons: Users may not be familiar with SQL

- **Alternative 2**: Use the current format of `n/NAME e/EMAIL …` (current)
    - Pros: Maintains consistency within the application
    - Cons: It is not as clear as SQL style syntax.

Alternative 1 would take the form of `find where name="John" and phone like "123" and not email="gmail.com"`. However, while SQL-style syntax is relatively clean and understandable, we decided not to implement this for the following reasons. The first reason was that SQL queries are meant to be used for large databases with thousands of columns, hence, the structure of the query must be clean to reduce the number of bugs made in the query. On the other hand, each `Entity` in Alfred only has a small number of fields (less than 6) and thus the current query format would still be understandable. Secondly, as mentioned above, users may not be familiar with SQL.

Hence, Alternative 2 was chosen as it would be more user friendly for users to the existing command format instead of memorizing a new format. Furthermore, implementing a new format would take more time as the new query format would have to be written and tested thoroughly, thus negatively impacting development velocity.

Aspect: Storing of the predicates

- **Alternative 1**: Store the Predicates in a central file (current)
    - Pros: Makes the predicates accessible to the whole application hence reducing code duplication
    - Cons: A large conditional statement would be needed in each Command class.
- **Alternative 2**: Store the Predicates within each `FindCommand` class
    - Pros: Each predicate is only used by one class, so this would have resulted in a better separation of concerns
    - Cons: A large conditional statement or hash map would still be needed to decide which predicates to use

Ultimately, we decided to use Alternative 1. Although most of the predicates are only used once, they are all unmodifiable functions that return a predicate in the form of a closure. Thus, the risks associated with unrestricted access and modification of an object are greatly reduced. Furthermore, predicates are also used in other parts of the codebase. Hence, it is a good idea to consolidate them in a single location. Thus, if another class in the codebase needed to implement another predicate, all it had to do would be to check this central file to see if a similar one was already written, as opposed to trawling though the entire codebase.

The large conditional statement was also not a major issue. This is because although it is large, the code within was understandable and this would have facilitated its extension, as compared to prematurely refactoring out into smaller submodules.