Tan Yuanhong - Project Portfolio

PROJECT: AlgoBase

Overview

AlgoBase is a desktop address book application used for teaching Software Engineering principles. The user interacts with it using a CLI, and it has a GUI created with JavaFX. It is written in Java, and has about 10 kLoC.

About this portfolio

This project portfolio details my individual contributions to the **AlgoBase** project. It includes a summary of the enhancements and other contributions I made throughout the duration of the project. Additionally, portions of my contribution to the User Guide and Developer Guide have also been included.

Summary of contributions

- **Major enhancement**: Implemented the problem searching and sorting feature.
  - What it does: allows the user to filter problems by combining search constraints on most fields (e.g. problem names, sources, descriptions, difficulty, etc.), allows user to save and reuse some typical search rules and allows user to sort the search result.
  - Justification: To organize algorithmic questions, the first step is to find a proper subset of AlgoBase to either add to a training plan or solve the problems directly. The search feature is one of the most frequently used functionality in **AlgoBase**.
  - Highlights: This is a full-stack feature all the components of **AlgoBase** from `Logic` to `Model` to `Storage` to `UI`. It requires an in-depth analysis of the overall architecture of **AlgoBase** as the original implementation of `find` in AddressBook accepts only one search constraint (i.e. the name), but in **AlgoBase** we need to support combination of arbitrary (non-zero) number of search constraints.
  - Relevant pull requests: #64 #90 #94 #107
- **Minor enhancement**:
  - `rewind`
- **Code contributed**: Funtional and Test Code
- **Other contributions**:
  - Project management:
    - Managed releases `v1.3` (1 release) on GitHub

- Enhancements to existing features:
  - Re-implement the `help` command
- Documentation:
  - Explains how reserved words should work in **AlgoBase** (used as a reference in the development) #35
- Community:
  - PRs reviewed (with non-trivial review comments):
  - Contributed to forum discussions
  - Reported bugs and suggestions for other teams in the class

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.

Model component

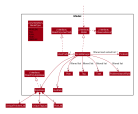

Figure 1. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user’s preferences.
- stores the AlgoBase data.
- exposes unmodifiable `ObservableList<Problem>`, `ObservableList<Tag>`, `ObservableList<Plan>`, `ObservableList<Task>`, `ObservableList<ProblemSearchRule>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- does not depend on any of the other three components.

We organize different data classes into packages (e.g. `Problem`), inside which we provide a collection class of that data object (e.g. `UniqueProblemList`) so that `AlgoBase` can manage these data objects without knowing the details of each data class.

Figure 2. Structure of the Problem Package

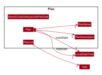
Figure 3. Structure of the Plan Package

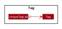
Figure 4. Structure of the Tag Package

Figure 5. Structure of the Task Package

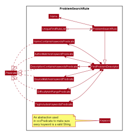
Figure 6. Structure of the ProblemSearchRule Package

Find Problem Feature

Since AlgoBase is a management tool for algorithmic questions, the search functionality is crucial to the user's experience with AlgoBase. For instance, the planning feature heavily relies on `find` command to determine the exact set of problems the user wants to include in a training plan.

This section will describe in detail the current implementation and design considerations of the find problem feature (i.e. search feature) of AlgoBase.

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

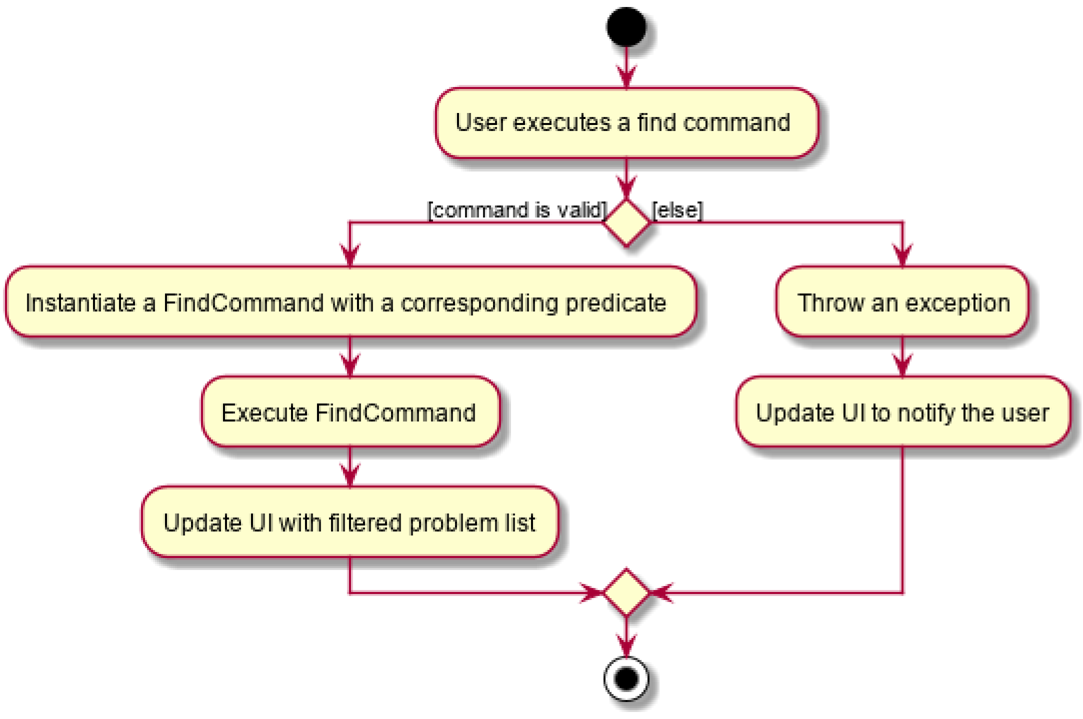
User executes a find command
[command is valid]
[else]
Instantiate a FindCommand with a corresponding predicate
Throw an exception
Execute FindCommand
Update UI to notify the user
Update UI with filtered problem list

Figure 7. Activity Diagram for the Execution of `find` Command

Current Implementation

The find problem feature mainly involves three parts:

1. validating and parsing user input
2. creating a filtering predicate from user’s search restrictions
3. update the displayed problem list with the filtering predicate.

The find problem feature is facilitated by the following classes:

- `FindProblemDescriptor`

  It stores predicates that are needed to describe a `FindCommand`

  Additionally, it implements the following operation(s):

  - `FindProblemDescriptor#isAnyFieldProvided()` - Determines if there is at least one search restriction included in this instance of `FindProblemDescriptor`.
  - `FindProblemDescriptor#equals()` - Two instances of `FindProblemDescriptor` are equal if and only if all of their predicates are equal.

- `FindCommandParser` It validates and parses user input to an instance of `FindCommand`.

  NOTE If the user provides difficulty range as one of the search restrictions, `FindCommandParser` expects the format `LOWER_BOUND <= difficulty <= UPPER_BOUND` while `LOWER_BOUND` and `UPPER_BOUND` are valid strings for doubles (i.e. parsable by `Double.parseDouble()`).

- `FindCommand`

  It creates and stores the `predicate` from an instance of `FindProblemDescriptor`. `predicate` is used to perform the filtering of the displayed problem list when the command is executed.

  `predicate` returns true only when the provided problem fulfills all restrictions described by the provided instance of `FindProblemDescriptor`.

  Additionally, it implements the following operation(s):

  - `FindCommand#execute()` - This method overrides `Command#execute()`. It filters problems in `filteredProblemList` in `model` with `predicate`.
  - `FindCommand#equals()` - Two instances of `FindCommand` are equal if and only if their `predicate` are equal.

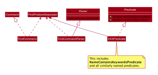
Command
FindProblemDescriptor
Parser
T parse(String userInput)
Predicate
boolean test(Problem problem)
boolean equals(Object other)
FindCommand
FindCommandParser
XXXPredicate
This includes NameContainsKeywordsPredicate and all similarly named predicates.

Figure 8. Class Diagram of the Find Feature

- Predicates that implements interface `Predicate<Problem>`

  These are classes that describes whether an instance of `Problem` is considered a match under a certain field with provided keyword(s).

- `NameContainsKeywordsPredicate`
  - It ignores case.
  - It returns true as long as one of the keywords appear in the name as a word. ("As a word" means the matching is done word by word. For instance, `hello` doesn't match `helloworld`.)
- `AuthorMatchesKeywordPredicate`
  - It is case sensitive and matches the entire author string (i.e. requires an exact match).
- `DescriptionContainsKeywordsPredicate`
  - It ignores case.
  - It returns true only when all of the keywords appear in the description as a word.
- `SourceMatchesKeywordPredicate`
  - It requires an exact match.
- `DifficultyIsInRangePredicate`
  - It matches problems with LOWER_BOUND <= difficulty <= UPPER_BOUND
- `TagIncludesKeywordsPredicate`
  - Each keyword will be considered as a tag, and two tags are considered equal only when their names are exactly the same.
  - It returns true when the provided tags are a subset of the tags of the provided problem.

Figure 9. Class Diagram for Predicates in the Find Feature

Given below is an example usage scenario and how the find problem mechanism behaves at each step.

Step 1. The user executes `find t/recursion diff/2.0-4.0` to find a problem with a tag "recursion" and difficulty between 2.0 and 4.0.

Step 2. `FindCommandParser` processes the user input and returns a `FindCommand` instance with the information of user's search restrictions.

NOTE	If no valid search restriction is provided by the user, `FindCommandParser` will throw a parsing exception, which is handled and displayed to the user.

Step 3. `LogicManager` invokes `execute()` method of the returned `FindCommand`. `FindCommand` updates the problem list with user's search restrictions.

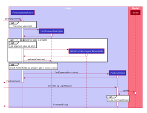

Figure 10. Sequence Diagram for the Execution of `find` Command

Design Considerations

Aspect: How to update the displayed problem list in the UI

- **Alternative 1 (current choice):** Let UI display problems in a `FilteredList<Problem>` and update the displayed problem by calling `setPredicate` on the `FilteredList`.
    - Pros: Provides good protection over unexpected changes on the displayed problem list.
    - Cons: Need to write a complex logic to generate one predicate out of multiple search constraints.
- **Alternative 2:** Let UI displays problems in an `ObservableList<Problem>` and update the list directly.
    - Pros: The implementation would be more straightforward as the logic can update the displayed list directly.
    - Cons: Leaves room for potential unexpected changes on the displayed problem list as the observable list is open to any kind of operation.

Aspect: How to deal with the case where no search restriction is provided (i.e. user types in `find` with no arguments given)

- **Alternative 1 (current choice):** Treat it as an exception and notify the user to provide at least one constraint.
    - Pros: Makes the meaning of `find` command clear - you can't search for problems without giving any conditions.
    - Cons: Has to check there is at least one predicate provided, making the implementation a bit more complicated.
- **Alternative 2:** Treat it as no restriction (i.e. `find` is equivalent to `list` in this case)
    - Pros: Easier implementation (if all predicates are always-true predicates, using `.and` method to chain them together would naturally result in an always-true predicate).
    - Cons: Confusing definition of a search function.

Aspect: How to make predicates optional (i.e. user doesn't have to provide restrictions for all searchable fields)

- **Alternative 1 (current choice):** Use `FindProblemDescriptor` in which the getter for the predicate returns `Optional<Predicate>`.
    - Pros: If the parser doesn't receive keyword(s) for a specific field, it simply doesn't call the descriptor's setter for that field. It doesn't need to deal with `null`, and `null` is dealt gracefully using `Optional.ofNullable(…)`
    - Cons: Rather troublesome implementation of the descriptor.
- **Altermative 2:** Store predicates in `FindProblemCommand` and check for not-provided predicates by comparing it with `null`.
    - Pros: More straightforward implementation.
    - Cons: If we are to add more predicates, it's more likely that we forget to check `null` value of

the new predicate.

Save Find Rules Feature

AlgoBase provides many ways to organizing your problems including tags and plans. However, both organizing features require persistent user involvement - if the user added a new problem belonging to a category, the user needs to manually assign a tag to the problem or add the problem to a plan. Since AlgoBase's `find` command enables the user to filter problems with great flexibility, we allow them to save certain find rules so that they can re-apply these rules to quickly locate problems of their need.

This section will describe in detail the current implementation and design considerations of the save find rules (or problem search rules) feature of AlgoBase.

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

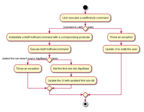
User executes a addfindrule command
[command is valid]
[else]
Instantiate a AddFindRuleCommand with a corresponding predicate
Throw an exception
Execute AddFindRuleCommand
Update UI to notify the user
[added find rule doesn't exist in AlgoBase]
[else]
Throw an exception
Add the find rule into AlgoBase
Update the UI with updated find rule list

Figure 11. Activity Diagram for the Execution of `addfindrule` Command

Current Implementation

The save find rules feature is facilitated by the following classes:

- `ProblemSearchRule`
  It stores both the `Name` of the find rule and all predicates included in this find rule. A `ProblemSearchRule` doesn't have to include all possible predicates as the user may not provide all of them. Missing predicates will be stored as `null` in this class.
- `UniqueFindRuleList`
  It stores the find rules and makes sure that every find rule in this list has a unique name.
  - `UniqueFindRuleList` stores a `ObservableList<ProblemSearchRule>` for UI purposes.

NOTE Except for `ProblemSearchRule`, we refer to these rules as `FindRule` in all other places. This is to prevent possible naming conflicts if AlgoBase is to support saving find rules on other items (e.g. Plans, etc.). `FindRule` corresponds to `FindCommand`. Thus, if you are to implement saving find plan rules, name them as `PlanSearchRule`, `AddFindPlanRuleCommand`, `UniqueFindPlanRuleList`, etc.

Under the category of save find rules feature, we have the following `Command` classes and their corresponding `Parser` classes:

- `AddFindRuleCommand`
- `DeleteFindRuleCommand`
- `ApplyCommand`
  It applies a problem-finding rule by specifying the index of the displayed find rule.

Since these commands share similar implementations, we will only take `AddFindRuleCommand` as an example since it's the most complicated one among the three.

Implementation of `addfindrule` feature

The `addfindrule` feature is facilitated by `AddFindRuleCommand` and `AddFindRuleCommandParser` class.

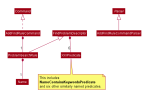

Figure 12. Class Diagram for Add Find Rule Feature

The sequence diagram below shows the high-level abstraction of how AlgoBase processes the request when user types in `addfindrule rule1 n/Sequences`:

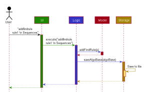

Figure 13. High-level Sequence Diagram for the Execution of `addfindrule rule1 n/Sequences`

The sequence diagram below illustrates the interaction between the `Logic` and `Model` component when executing `AddFindRuleCommand`. Notice that the constructor for `AddFindRuleCommand` requires `Name` to be non-null and accepts null values for other predicates. Thus if the predicate is not present in the arguments, `AddFindRuleCommandParser` will pass null to the constructor of `AddFindRuleCommand`.

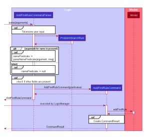

Figure 14. Sequence Diagram for the Execution of `addfindrule` *Command*

Design Considerations

Aspect: To implement `ProblemSearchRule` as a subclass of `FindProblemDescriptor` or implement it as a immutable concrete class.

Since AlgoBase is forked from AddressBook 3, it also inherits AB3's design choice on all data classes - they are all immutable classes with all fields being `final`. However, `ProblemSearchRule` is essentially saving the information of a command input, where the user may provide any number of predicates as the argument. We implement mutable `FindProblemDescriptor` to accommodate variable user inputs, now we have to consider whether to keep `ProblemSearchRule` immutable or not.

- **Alternative 1 (current choice):** `ProblemSearchRule` extends `FindProblemDescriptor` with an additional field `name`
  - Pros: Drastically reduces the amount of duplicate code as `ProblemSearchRule` shares most fields with `FindProblemDescriptor`
  - Cons: `ProblemSearchRule` as a data class is no longer immutable. We have to be careful not to call any setters it inherits from `FindProblemDescriptor`.
- **Alternative 2:** `ProblemSearchRule` as an individual class with immutable fields.
  - Pros: Provides good protection over unexpected changes to the data fields.

    - Cons: Lots of repeated code.