Nicholas Novakovic - Project Portfolio

PROJECT: El Duque

Overview

El Duque is a task manager application targeting computer science students. The user interacts with it using a CLI. It is written in Java, and has about 10 kLoC.

Summary of contributions

- **Enhancement**: added **the ability to undo previous commands**
  - What it does: allows the user to undo all previous commands one at a time.
  - 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.
- **Enhancement**: added **the ability to filter tasks**
  - What it does: allows the user to organise tasks based on user-defined categories.
  - Justification: This feature improves the product significantly because a user can now choose to view tasks from specific categories. This is especially helpful considering our target user audience is computer science students who will have many tasks of varying categories to do. This will help them better organise their tasks.
  - Highlights: This enhancement affects existing tasks, 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.
- **Code contributed**: [Functional code] [Test code] *{TODO: give links to collated code files}*
- **Other contributions**:
  - Project management:
    - Managed all JAR releases on GitHub
  - Enhancements to existing features:
    - Updated the formatting of how the task list is printed to the user, increasing the readability and improving the user experience (Pull requests {to be added})
  - Documentation:
    - Converted the entire DeveloperGuide from a Google Doc to an AsciiDoc to improve the

formatting

- TODO → convert UG similarly

- Tools:
  - Integrated a third party library (Gson) to the project

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.

{TODO - convert UG to adoc for easy conversion to here}

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.

Logic component

[LogicClassDiagram] | *LogicClassDiagram.png*

Figure 1. Structure of the Logic Component

1. `Logic` uses the `DuqueParser` to parse the user command.
2. Depending on the user command, a different parser will be created to handle the command.
3. This results in a `Command` object which is executed by `El Duque`.
4. The command execution can affect the `Model` (e.g. adding a `Task`).
5. The result of the command execution, which may affect the `Model`, will be passed back to the `Ui` to display confirmation of the successful execution to the user.

Filtering

Implementation

The filter mechanism involves attaching filters to tasks or events which is facilitated by `Parser` and `AddCommand`.

`Parser` reads user input and if the first word in the user input starts with a `-` character, `Parser` will identify the first word as a filter and carry out the remaining command using the filter. This enables the following user commands:

- `[<filter>] task/event [<description>] [<labels>]` — Creates a new task or event with relevant description and modifications as well as a filter for the task.
- `[<filter>] list [<labels>]` — List all tasks and events with the relevant filter and labels

- `[<filter>] edit [<index>/<description>] [<labels>]` — Edit a task or event seen in the relevant filtered list by its index in that list or by its description.

Given below is an example usage scenario and how the filter mechanism behaves at each step:

```
-CS2113 task DG submission -d 2 -t 251019 2359
-CS2113 list
-CS2113 edit 1
```

1. The user launches the application for the first time. The `Duke` main class will be initialized, and the `Ui` class will prompt the user to key in input.
2. The user executes `-CS2113 task DG submission -d 2 -t 251019 2359` command to add a new task called `DG submission` which will have a `filter` `CS2113`, a `duration` of `2 hours` to complete, and a deadline at `25/10/2019 23:59`.

NOTE	If the user's system somehow crashes after executing the above command, the new task entry will still be saved into the JSON storage file and can be recovered on the next launch of the application.

3. The user executes `-CS2113 list` to view all tasks and events associated with `CS2113`.
4. The user now decides that setting the duration of `DG submission` to be only 2 hours was a mistake, and decides to increase the duration needed by executing `-CS2113 edit 1 -d 4`. The `edit` command will call `EditCommand`, which will search for the corresponding task in the `TaskList`, updating whatever values the user has input, in this case updating the `duration` to `4 hours`.

Design Considerations

Aspect: How filter works

- **Alternative 1 (current choice):** Use an `Optional<String>` attribute within `Task` to keep track of what filter each `Task` has.
    - Pros: Easy to implement.
    - Cons: May have performance issues in terms of speed when calling list because a new list must be filtered from all existing tasks within the current `TaskList`.
- **Alternative 2:** When a new `filter` is created, create a new `TaskList` specific to that `filter` to store those tasks.
    - Pros: Will be faster to show filtered list to the user
    - Cons: Must change implementation of `TaskList`, `TaskListPrinter`, `AddCommand`, `EditCommand` to facilitate this. We must ensure that the implementation of each individual command is correct.

Undo

Implementation

The undo mechanism is facilitated by `Duke`, `UndoStack`, and `UndoCommand`. `UndoStack` stores the current undo history internally as a `java.util.Stack` object. `UndoStack` only stores undo information for when the user executes a command that we consider "undo-able". Undo-able commands include the following: `AddCommand`, `DeleteCommand`, `EditCommand`, and `DoneCommand`. Undo information is actually mirror `Command` classes that will do the opposite of what the current command has done. For example, the mirror of `AddCommand` is `DeleteCommand`. This storing of undo information is facilitated by `Duke`. `Duke` calls the `savePrevState()` method of these commands to create respective mirror classes.

When `UndoCommand` is executed, it will check whether `UndoStack` contains any commands using the `UndoStack` method `isEmpty()`. If `UndoStack` is not empty, `UndoCommand` will call the `UndoStack` method `retrieveRecent()` to obtain the most recent undo-able command that the user called. `UndoCommand` will then execute that command, undoing the user's most recent undo-able command.

Given below is an example usage scenario and how the undo mechanism behaves at each stage:

```
task mistake
undo
```

1. The user launches the application for the first time. The `Duke` main class will be initialized, and the `Ui` class will prompt the user to key in input.
2. The user executes `task mistake` command to add a new task called `mistake`. `AddCommand` will be executed to create the task and add it to the current `TaskList`. `Duke` will call the method `savePrevState()` on the `AddCommand` created, which will store a mirror `DeleteCommand` that corresponds to the newly added task into the `UndoStack`.
3. The user realises that adding that task was a mistake and wants to undo his action of adding the task. The user now executes `undo` command to undo his previous action of adding the mistake task.
4. `UndoCommand` will be created and executed. `UndoCommand` calls the `isEmpty()` method of `UndoStack` and realises that there is at least one command that can be undone. `UndoCommand` then calls `retrieveRecent()` method of `UndoStack` to obtain the most recent undo-able command's mirror command. `UndoCommand` will execute this mirror command and undo the recent adding of the task `mistake`.

Design Considerations

Aspect: How undo works

- **Alternative 1 (current choice):** Use a `java.util.Stack` to store mirror commands that facilitate the undoing of undo-able commands.
    - Pros: Easy to implement, fast, saves space.
    - Cons: Difficult to concurrently implement a redo feature.

- **Alternative 2:** Use a java.util.List to store each state of the TaskList whenever it changes regardless of whether calling Undo/Redo.
    - Pros: Able to efficiently execute undo/redo.
    - Cons: Takes up more space which might slow down the program overall or even cause an eventual `Memory Limit Exceeded` exception.

Appendix A: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be able to hold up to 1000 tasks without a noticeable sluggishness in performance for typical usage.
3. 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.

Appendix B: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Appendix C: 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 CLI with a welcome message. The window size may not be optimum.
2. Shutdown
    a. Enter the command `bye`
    Expected: A goodbye message will be shown and the app will exit.

Adding a task

1. Adding a new simple task
    a. Test case: `task example`

Expected: New task created called example which can be viewed by calling `list`. This undone task will have no filter, no datetime, no recurrence, no duration, and a low priority.

2. Adding a new complex task
    a. Test case: `-test task example -d 7 -t today -r daily`
    Expected: New task created called example which can be viewed by calling `list`. This undone task will have a filter `test`, the current local datetime, a daily recurrence, a 7 hour duration to complete, and a low priority.

Deleting a task

1. Deleting a person while all tasks are listed
    a. Prerequisites: List all tasks using the `list` command. At least one task in list.
    b. Test case: `delete 1`
    Expected: First task is deleted from the list. Details of the deleted task shown in the status message.
    c. Test case: `delete 0`
    Expected: No task 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) _ Expected: Similar to previous.
2. Deleting a person while filtered tasks are listed
    a. Prerequisites: List filtered tasks using the `<filter> list` command. At least one task in filtered list.
    b. Test case: `<filter> delete 1`
    Expected: First task is deleted from the `<filter> list`. Details of the deleted task shown in the status message.
    c. Test case: `<filter> delete 0`
    Expected: No task is deleted. Error details shown in the status message. Status bar remains the same.
    d. Other incorrect delete commands to try: `<filter> delete`, `<filter> delete x` (where x is larger than the filtered list size) _ Expected: Similar to previous.