

Introduction

Assignment Goals

After completing this group assignment, each student is expected to be able to Learning Outcomes

- Make up *user stories* for how a user might interact with a program.
- Design a `UserCommand` object's **public** interface. (Note: not a Java **interface**)
- Design simple, single-responsibility *methods*

Procedure

1. You have been hired to make a *command-driven* program for a large client. You know that if you do well, there will be much more work for them *and* that they will appreciate a unified input scheme across multiple programs.

- (a) Before reading further, with your partner, describe in two to four sentences, what you think of as a *command-driven* program. What, at a high-level, is the user experience?
- (b) Most *interactive* programs have the same structure (from word-processor to video game):

```
while (!done)
    check for user input
    update program state
    display program state
```

In a word-processor, for example, we check for keyboard/mouse activity. If there is some, use it to add/remove characters. Perhaps add “misspelled” squiggles or highlight matching words (update the state). Then draw the screen. This happens multiple times per second.

Consider a simple, mobile, arcade-style game. You and your partner describe the three parts of this loop, again in a few sentences.

- (c) Consider a program that runs at a much more sedate pace: a to-do list manager, **ToDa!**. check **for** user input becomes `get user input` because **ToDa!** only updates state or display in response to user *commands*.
 - i. Describe, as above, the parts of the loop for **ToDa!**. Pay particular attention to the contrast with the game/word-processor.
 - ii. List *all* the commands you can think of for a to-do list manager. Assume it is run in a terminal on the desktop.
2. Take a slightly different view of **ToDa!**: describe how a user, new to the application, would interact with it to enter two (2) to-do items, list all the items, and then remove one of them.

Entering an item or identifying an item for deletion is beyond the scope of this design session. Pay attention to how we give the application a *command*, like `edit` which would then (in some other code) identify the item and would then (in still other code) find the item and would then (in other other code) let the user edit the data.

This is probably a sketch of an interactive session with the non-existent software.
3. Look at your sketch: How did the naïve user know what commands to give the system to do what they wanted? How could you (easily) support helping the user to discover valid commands? Does it make sense to have multiple help messages displayed in different contexts?
4. Put back on your programmer's hat: Given your interaction above, including the need for command discovery, what does the **public** interface (*set of public methods*) of a **reusable** `GetUserCommand` class have.

What parts of getting the `edit` command from the user can be separated from the processing of to-do items?

After you have written a response here, ask Dr. Ladd for page 2.

Authored by an LLM:

Objective:

In this exercise, you will design a reusable command processor that reads valid commands and their corresponding help text from a file, provides general and specific help information, ensures that no command is a prefix of any other command, and accepts unique prefixes. The scanner used for reading commands will be injected as a dependency.

Design Requirements:

1. Read valid commands and their corresponding help texts from a file.
2. Implement the '`!`' or '`? <command>`' mechanism to show general or command-specific help information.
3. Loop until a valid command is provided by the scanner.
4. Ensure that no command is a prefix of any other command.
5. Allow unique prefix selection for commands.
6. Inject an instance of '`java.util.Scanner`' to read user input.

Steps:

1. Create a class named '`CommandProcessor`'. This class should have a constructor that accepts an instance of '`java.util.Scanner`' as an argument.
2. Define a method named '`loadCommandsFromFile`' in the '`CommandProcessor`' class. This method should load valid commands and their corresponding help texts from a file, storing them in a suitable data structure (e.g., a '`Map<String, String>`').
3. Implement the '`!`' or '`? <command>`' mechanism by defining a method named '`showHelp`'. This method should check if the user input is either '`!`' or '`? <command>`', and display the corresponding help text based on the input.
4. Define a method named '`processCommand`'. This method should loop until a valid command is provided by the scanner. Inside this method, check for unique prefixes to ensure that no command is a prefix of any other command. If the user input does not match any existing command, display an error message and prompt the user to try again.
5. Inject an instance of '`java.util.Scanner`' into the '`CommandProcessor`' class through its constructor. This scanner will be used to read user input.
6. Test the '`CommandProcessor`' class by creating a simple application that prompts users for commands and provides help information as needed.