Design Analysis

Command API:

- **Defining Commands**:
  - The Command class allows developers to define commands with any number of positional and named arguments.
  - Arguments can be added to commands by specifying their name, type, and whether they are required or optional.
- **Extracting Values**:
  - Parsed arguments are extracted using the Command.parse() method, which returns a map of argument names to their parsed values.
- **Handling Arguments**:
  - The library effectively handles positional and named arguments by distinguishing between the two during command registration and parsing. The separation between named and positional arguments is clear, and the library enforces the correct order of positional arguments.

Argument API:

- **Defining Arguments**:
  - Arguments are defined using the Argument class, which takes a name, a parser, and flags indicating whether the argument is required or named.
- **Parsing and Validation**:
  - Parsers (ArgumentParser implementations) handle the conversion of strings into the appropriate data types.
  - RangeParser and ChoiceParser were implemented to handle specific validation requirements, like ensuring values are within a valid range or among specific choices.
- **Handling Required and Optional Arguments**:
  - Required arguments are enforced during command parsing, and an ArgumentParseException is thrown if a required argument is missing.
  - Optional arguments are managed by checking if a value was provided and applying default values if necessary.

Error Handling:

- **Representation of Errors**:

    - Errors are represented using the ArgumentParseException class for parsing-related issues and Lexer.ParseException for tokenization errors.
- **Separation of Errors**:
    - Developer errors (e.g., defining commands incorrectly) are separated from end-user errors (e.g., providing invalid input) by throwing different exceptions and providing descriptive messages for each type of error.
- **Detail of Errors**:
    - Error messages are designed to provide enough detail to help users understand what went wrong and how to correct it. For example, range validation errors include the expected range, and choice validation errors include the list of valid options.

Individual Design Analysis

Summary of Individual Work: Hoang Le Duc

- Implemented the core classes (RangeParser, ChoiceParser, LocalDateParser, Argument, Command, and CommandManager).
- Updated Main.java to integrate the new command and argument handling.
- Added execution logic for each command and handled unsupported features with manual validation.

Well-Designed Part of the Implementation:

- **Command and Argument System**:
    - The Command and Argument classes are well-designed as they encapsulate command-specific logic and argument properties, respectively. This separation of concerns makes the code more modular and easier to extend.
    - The use of parsers for arguments allows for clean validation and type conversion, adhering to the principle of **Single Responsibility**—each parser is responsible for validating and converting its specific type.
    - By having a clear separation between command definitions (Command) and command management (CommandManager), the design follows **Encapsulation** and **Modular Design**, making it easier to register and manage commands.

Poorly Designed Part of the Implementation:

- **Handling Optional Arguments**:
    - The implementation of optional arguments and default values currently requires manual handling within each command's execution logic. This is not ideal, as it

makes the code less reusable and increases the complexity of command definitions.
    - This part of the implementation lacks a consistent way of handling defaults, leading to potential duplication of logic. It would be better to incorporate default handling directly into the Argument or Command class.

Most Risky Part of the Project:

- **Handling Complex Argument Parsing**:
    - Extending the library to support complex argument parsing, such as multiple optional arguments or subcommands, introduces the most risk. Managing how arguments are parsed, matched, and validated when they have different types, orders, and defaults will be challenging.
- **Questions/Concerns**:
    - How should skipped arguments be handled when multiple optional arguments are provided? Should a specific syntax be enforced?
    - How can subcommands be integrated into the existing command structure without significantly increasing the complexity of the Command class?

Summary of Individual Work: Calvin Perez

- Updated Main.java for error handling

Well-Designed Part of the Implementation:

- **Error handling within command execution**
    - I separated the code that checked for additional arguments within the command definition and execution. Including this in the execution would help separate error handling in its own area.

Poorly Designed Part of the Implementation:

- **Handling additional arguments in command definition**
    - I added additional arguments to check if there were any arguments that didn't belong in the command that was called. This however, seems less robust and potentially has improvement by implementing this in another area.

Most Risky Part of the Project:

- **Handling different arguments**
    - When calling other commands I think there could be problems with certain arguments being invalid, as in I think that there are certain edge cases we didn't consider while writing the code.
- **Questions/Concerns**:

- How should we standardize the handling of optional arguments with default values to ensure that the logic is reusable across different commands?

Team Design Analysis

Differing Opinions:

- One potential area for differing opinions is how to handle optional arguments. Some might prefer to handle optional arguments within the Argument class, while others might favor keeping that logic within the command execution method. Comparing these justifications, a design where Argument handles default values is more maintainable and reusable, but it could add complexity to the argument parsing logic.

Handling Extraneous Arguments:

- Originally, there was no check for extraneous arguments. After encountering some unexpected behavior, the design was changed to include validation for additional arguments during command parsing. This change improved the robustness of the library by ensuring that commands do not accept unintended arguments, which could lead to incorrect program behavior.

Advantages and Disadvantages of the Current Design:

- **Separation of Command, Argument, and Parser**:
  - **Advantages**:
    - **Modularity**: The separation allows each component to focus on a single responsibility, making the system easier to understand, maintain, and extend.
    - **Reusability**: Parsers can be reused across multiple commands, reducing duplication.
  - **Disadvantages**:
    - **Complexity in Integration**: Separating commands, arguments, and parsers can make the process of defining commands more verbose and complex, especially for new developers who may not understand how these pieces interact.

Addressing Pain Points from argparse/argparse4j:

- The current design addresses some pain points of argparse and argparse4j:
  - **Custom Parsers**: The use of custom parsers (CustomParser) makes it easier to extend the library to support new types without modifying the core parsing logic.

- **Error Messages**: Detailed error messages that explain what went wrong (e.g., invalid range or invalid choice) provide a better user experience compared to the generic errors sometimes seen in argparse.