Report on a Web App for Viewing and Editing Note Collections

1. Features of the submitted program

The Java program implements a Note Collection System for managing and editing notes. It supports almost all features outlined in the specification and can be run using the command `mvn clean compile package exec:exec`.

The system provides a graphical user interface (GUI) for viewing and categorizing notes. Each note has a title and contains textual content, which can be edited using a subset of Markdown syntax. Supported formatting options include collapsed spaces, bulleted and numbered lists, code blocks, quote blocks, as well as bold, italic and monospaced text. HTML escape characters are used to prevent injection attacks.

Notes are organized into indices, with all notes initially placed in a default "Root" index. Indices can be nested, enabling the creation of sub-indices at multiple levels. Notes within an index can be sorted alphabetically, in reverse alphabetical order, or by length, which is calculated as the number of non-whitespace characters. All notes and indices, other than the "Root" index, can be created, deleted and renamed in line with data validation rules. All data are stored in a `notesData.json` file, which is automatically updated whenever changes are made.

2. Evaluation of design decisions and programming processes

To ensure maintainability and extensibility, the project's architecture closely follows the model-view-controller (MVC) design pattern. The discussion below details how each of these components were designed and realized in accordance with the principles of object-oriented thinking.

2.1. View: Java Server Pages (JSPs)

Alongside the `style.css` stylesheet, each page's appearance and layout are defined using JSP files. Despite having a combination of HTML elements and Java code, each JSP file acts solely as a "view" of the program and does not perform any data processing other than sending HTTP requests and visualizing their responses.

2.2. Model: Java classes

The program's "model" consists of Java classes that are independent from the GUI.

One such class is `Model`, which handles core operations such as ensuring consistency between program data and JSON data. It can only be instantiated by the `ModelFactory`, which imposes a singleton pattern and ensures that only one `Model` object is present throughout program execution. A similar approach is employed by

`ObjectMapperFactory`, which facilitates the reuse of Jackson's `ObjectMapper` by preventing multiple instances from coexisting at the same time.

The process of determining which additional classes to implement began with the selection of important nouns from the requirement specification. This gave rise to the classes `Note` and `Index`, which provide abstractions for their respective namesakes. The discovery that `Note` and `Index` share many common methods revealed the need to follow the Don't Repeat Yourself (DRY) principle, leading to the creation of the abstract `IndexEntry` class, from which both classes inherit. The `IndexEntryPath` class was also created to aid navigation between indices at different levels.

To enhance modularity and separation of concerns, it was decided that operations like `markdownToHTML(String)` and `sortAtoZ(ArrayList<Note>)` should not be directly incorporated into `Note` or `Index`. This motivated the creation of classes `TextFormatter` and `NoteSorter`, where these methods are declared as static. The enum `NoteOrdering` was defined to increase the readability of `NoteSorter`.

Lastly, encapsulation is enforced by declaring all instance variables as private. Public accessors are defined only when necessary, whereas other instance methods are declared private or protected whenever possible, thus limiting their scope.

2.3. Controller: HTTP servlets

The model and view components are linked via HTTP servlets, which act as the program's controller by receiving HTTP requests from the client and generating responses based on data from the model.

All servlets extend the abstract class `AbstractHttpServlet`, which handles GET and POST requests. This leverages polymorphism, since the concrete classes extending it can override the abstract method `respondAndGetJSP(…)` with different implementations. Furthermore, this reduces code duplication: the code for invoking JSPs, for example, appears only in `AbstractHttpServlet` but not in any of its subclasses.

2.4. Overall quality

Given its thorough use of abstraction, encapsulation and inheritance, the program is thought to serve as a well-rounded demonstration of key object-oriented design principles. The program adheres strictly to the MVC design pattern, which lends itself to better modularity, improved testability and streamlined maintenance.

Nevertheless, certain aspects of the program design present room for improvement. This includes furthering the separation of concerns by migrating methods for manipulating JSON data (e.g. `updateNoteJsonNode(…)`) from `Model` to a separate class that utilizes `ObjectMapper`; and making more extensive use of inheritance with regard to the ten servlet classes, which currently all directly extend one abstract class.