DWA_07.4 Knowledge Check_DWA7

---

1. Which were the three best abstractions, and why?

1. Abstraction: createPreview Function
    - Purpose: This function encapsulates the process of creating a preview element for a book with its image, title, and author.

    - Benefits: It abstracts away the HTML structure and styling details needed for a book preview, making the code more readable and maintainable.

    - Reusability: This abstraction can be reused whenever you need to display a book preview, such as in search results or other book lists.

    - Simplicity: It simplifies the process of creating a consistent book preview across the application.

2. Abstraction: populateBookItems Function
    - Purpose: This function encapsulates the logic for populating a list of book previews within a specified range.

    - Benefits: It abstracts the process of iterating through the filtered books, creating previews, and appending them to the HTML. It promotes modularity and clarity by separating concerns.

    - Reusability: This function can be reused whenever you need to populate a list of book previews, ensuring consistency in how books are displayed.

    - Simplicity: It simplifies the code that deals with adding book previews to the UI, making it easier to understand.

3. Abstraction: updateShowMoreButton Function
    - Purpose: This function abstracts the logic for updating the "Show more" button based on the number of remaining books.

    - Benefits: It encapsulates the calculation of remaining books and updating button content and state. This abstraction enhances code readability and separates UI-related logic.

- Reusability: This function can be used whenever you need to update buttons with remaining book counts, providing a consistent way of handling this UI element.

- Flexibility: It makes it easier to adjust the button's appearance or behavior in one place.

These three abstractions stand out for their clarity, reusability, encapsulation, and simplicity in handling specific aspects of the UI and logic. They contribute to maintaining a well-organized and maintainable codebase.

---

2. Which were the three worst abstractions, and why?

1. Abstraction: toggleListDialog Function
    - Issue: While the function toggles the visibility of a dialog, its name doesn't clearly convey its purpose or intent.

    - Improvement: Renaming the function to something more descriptive, like toggleBookDialogVisibility, would make its purpose clearer and improve code readability.

2. Abstraction: handleSearchFormSubmit Function
    - Issue: This function handles several different responsibilities, including form submission, filtering books, updating UI elements, showing messages, and more.

    - Improvement: Splitting this function into smaller, focused functions (e.g., filterBooks, updateUI, showMessage) would improve modularity and make the code more maintainable.

3. Abstraction: handleSettingFormSubmit Function
    - Issue: Similar to the previous example, this function handles multiple tasks, such as form submission, setting color properties, and closing the settings overlay.

- Improvement: Separating the concerns into distinct functions (e.g., setThemeColors, closeSettingsOverlay) would make the code more modular and easier to follow.

In these cases, the abstractions might be considered less effective due to unclear naming, mixed responsibilities, and lack of modularity. By addressing these issues, you can improve the overall quality and maintainability of your codebase.

---

3. How can The three worst abstractions be improved via SOLID principles.

1. Abstraction: toggleListDialog Function
    - Issue: Unclear naming and potential mixing of responsibilities.
    - Improvement:
        - Single Responsibility Principle (SRP): Rename the function to toggleBookDialogVisibility to clearly indicate its purpose.
        - Open/Closed Principle (OCP): Consider using a strategy pattern to handle different dialog types (e.g., book dialog, search dialog, settings dialog). This way, you can extend the functionality without modifying the existing code.

2. Abstraction: handleSearchFormSubmit Function
    - Issue: Mixing responsibilities, including form submission, filtering books, updating UI, and showing messages.
    - Improvement:
        - Single Responsibility Principle (SRP): Break down the function into smaller functions, each with a single responsibility:
            - filterBooks - Responsible for applying filters to the list of books.
            - updateUI - Handles updating UI elements based on filtered results.
            - showMessage - Manages displaying a message based on the filtered book count.
        - Dependency Inversion Principle (DIP): Introduce interfaces or abstractions for interacting with UI elements. Pass these abstractions to the smaller functions to decouple them from specific UI implementations.

3. Abstraction: handleSettingFormSubmit Function
    - Issue: Mixing responsibilities, including form submission, setting color properties, and closing the settings overlay.
    - Improvement:
        - Single Responsibility Principle (SRP): Separate the function into distinct responsibilities:
            - setThemeColors - Handles setting CSS custom properties for theme colors.
            - closeSettingsOverlay - Manages the visibility of the settings overlay.
        - Interface Segregation Principle (ISP): If applicable, define separate interfaces for setting theme colors and managing
            - overlays, to ensure that each function adheres to a specific set of responsibilities.

By applying SOLID principles, you'll be able to improve the clarity, modularity, and maintainability of your code. Each function will have a clear responsibility, making the codebase easier to understand and extend. Additionally, the principles help in reducing the impact of changes in one part of the code on other parts, promoting a more flexible and robust design.
