DWA_07.4 Knowledge Check_DWA7

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

The BookPreviewsManager - This class is responsible for managing and displaying book previews on the page. It abstracts the logic related to populating book previews, loading more previews, and interacting with the DOM elements. By encapsulating these functionalities within a dedicated class, the code becomes more modular and easier to understand. It separates the concerns of handling book previews from other parts of the application, making the codebase more organized and maintainable.

The BookDetailsManager - This class handles the display and hiding of book details modal. It abstracts away the implementation details of showing and hiding the modal, providing a cleaner and more focused interface. This abstraction helps in reducing redundancy and makes it easier to manage the interaction with the book details modal throughout the code. It encapsulates the behavior associated with book details, making the code more readable and easier to update in the future.

The SearchFilterManager - The SearchFilterManager class abstracts the logic related to applying search and filter criteria to the book previews. By centralizing this functionality, the codebase becomes more cohesive and avoids scattering filtering-related code throughout the application. This abstraction promotes code reuse since the filtering logic is encapsulated within a single class. It enhances the maintainability of the code by making it easier to update or extend the filtering behavior in the future.

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

CSS Theme Management - In the original code, the ThemeManager class is responsible for applying themes to the interface by directly manipulating the CSS variables. While abstracting the theme application is a good idea, the implementation could be improved. Direct manipulation of CSS variables in a JavaScript class might not be the most maintainable approach. It could be better to abstract the theme-related CSS into separate stylesheet files, and the ThemeManager class could then toggle class names or attributes to switch between themes. This would separate the visual styling from the JavaScript logic and provide a cleaner way to manage themes.

Event Handling and DOM Selection: Throughout the code, there is still a considerable amount of event handling and DOM element selection directly within methods. While the classes abstract some behaviors, they still interact with the DOM and handle events. Ideally, the abstraction could be taken further by encapsulating these interactions within dedicated classes or functions. A more effective approach could be to create separate modules responsible for event handling and DOM manipulation, allowing the core classes to focus on their specific responsibilities.

Code Duplication: While the refactored code addresses some code duplication, there are still instances of repetitive code. For example, the logic for creating and appending book previews appears in both the BookPreviewsManager and the event listeners for the "Show more" button. This duplication could be reduced by creating a shared function or method that handles the creation and appending of previews. By centralizing this logic, changes or updates would only need to be made in one place, reducing the risk of inconsistencies.

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

CSS Theme Management: Single Responsibility Principle (SRP): The responsibility of the ThemeManager class should be solely focused on managing themes. To adhere to SRP, separate the theme-related CSS styles into dedicated stylesheet files. Then, create a ThemeManager class that toggles classes or attributes on the HTML elements to switch themes. This way, the ThemeManager class is responsible only for changing themes, and the visual styling is handled in a separate file.

Event Handling and DOM Selection: Single Responsibility Principle (SRP): The core classes like BookPreviewsManager, BookDetailsManager, and SearchFilterManager should not be directly responsible for DOM manipulation and event handling. Create separate modules or classes for handling event listeners and DOM interactions. This separation adheres to the SRP and ensures that each class has a single responsibility.

Code Duplication: Don't Repeat Yourself (DRY) - To address code duplication, create shared functions or methods for common operations. For example, the logic for creating and appending book previews could be placed in a single method that both the BookPreviewsManager and the "Show more" button event listener can call. This adheres to the DRY principle by promoting code reuse and reducing redundancy. Open/Closed Principle (OCP): When creating shared methods for common operations, design them to be open for extension but closed for modification. This means that if you need to make changes or improvements to these methods in the future, you can do so without modifying the existing code. This principle encourages a modular approach that minimizes the impact of changes on the rest of the codebase.
