DWA_07.4 Knowledge Check_DWA7

---

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

The BookPreviewsManager:

This class manages and displays book previews on the page. It handles the logic for populating book previews, loading more previews, and interacting with DOM elements. By encapsulating these functions in a single class, I find the code becomes more modular and easier to understand. This separation helps keep the codebase more organized and maintainable by isolating book preview management from other parts of the application.

The BookDetailsManager:

This class manages the display and hiding of the book details modal. It abstracts the implementation details of showing and hiding the modal, providing a cleaner and more focused interface. This abstraction reduces redundancy and simplifies managing interactions with the book details modal throughout the code. By encapsulating the behavior associated with book details, I find the code more readable and easier to update in the future.

The SearchFilterManager:

This class handles the logic for applying search and filter criteria to book previews. By centralizing this functionality, I notice the codebase becomes more cohesive and avoids scattering filtering-related code. This abstraction promotes code reuse by encapsulating the filtering logic in a single class, which enhances maintainability and makes it easier to update or extend filtering behavior in the future.

---

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

CSS Theme Management:

The ThemeManager class applies themes by directly manipulating CSS variables. While abstracting theme application is a good idea, I believe the implementation could be improved. Directly manipulating CSS variables in JavaScript might not be the most maintainable approach. I think it would be better to abstract theme-related CSS into separate stylesheet files and have the ThemeManager toggle class names or attributes to switch themes. This would separate visual styling from JavaScript logic, providing a cleaner way to manage themes.

Event Handling and DOM Selection:

I noticed that the code still contains a significant amount of event handling and DOM element selection within methods. Although some behaviors are abstracted into classes, they still interact with the DOM and handle events. A more effective approach would be to encapsulate these interactions within dedicated classes or functions. Creating separate modules for event handling and DOM manipulation would allow the core classes to focus on their specific responsibilities.

Code Duplication:

While some code duplication has been addressed, I still see 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. Centralizing this logic would ensure that changes or updates need only 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 ThemeManager class should focus solely on managing themes. To adhere to SRP, I would separate theme-related CSS styles into dedicated stylesheet files. Then, I would create a ThemeManager class that toggles classes or attributes on HTML elements to switch themes. This approach ensures the ThemeManager class is responsible only for changing themes, while visual styling is handled separately.

Event Handling and DOM Selection:

Single Responsibility Principle (SRP): Core classes like BookPreviewsManager, BookDetailsManager, and SearchFilterManager should not handle DOM manipulation and event handling directly. I would create separate modules or classes for handling event listeners and DOM interactions. This separation adheres to SRP and ensures each class has a single responsibility.

Code Duplication:

Don't Repeat Yourself (DRY): To address code duplication, I would create shared functions or methods for common operations. For example, I would place the logic for creating and appending book previews in a single method that both the BookPreviewsManager and the "Show more" button event listener can call. This approach adheres to the DRY principle by promoting code reuse and reducing redundancy.

Open/Closed Principle (OCP): When creating shared methods for common operations, I would design them to be open for extension but closed for modification. This means changes or improvements to these methods can be made without modifying the existing code. This principle encourages a modular approach that minimizes the impact of changes on the rest of the codebase.