DWA_07.4 Knowledge Check_DWA7

---

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

- **`createButtonElement`:** This function eliminates the logic for creating a button element based on the properties of a book object. It takes care of setting the appropriate attributes, classes, and inner HTML of the button. This abstraction is valuable because it separates the creation of the button element from the surrounding context. It promotes code reuse by allowing the function to be used in different parts of the application where button elements need to be created for books.

- **`createOptions` and `createOptionElement`**: These functions are responsible for creating the options for the genre and author select dropdowns. They take care of creating option elements based on the provided data and adding them to the respective container in the DOM. By abstracting this functionality into separate functions, the code becomes more modular and easier to understand. It promotes code reuse and allows for easy extension in case new options need to be added.

- **Event handler functions**: The code defines various event handler functions, such as `handleSearchCancel`, `handleSettingsCancel`, `handleHeaderSearchClick`, etc., which handle specific user interactions. These functions abstract away the implementation details of handling these events and provide a higher-level interface. They encapsulate the behavior associated with specific user actions and allow for easy maintenance and modification of event handling logic. They contribute to the separation of concerns and promote code organization.

---

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

- **`handleListButtonClick`**: This event handler function is responsible for handling the "Show more" button click. While it performs the task of loading more book items onto the list, it also takes care of updating the button's text and remaining book count. This function could be improved by separating the responsibilities more clearly. Instead, it could focus solely on loading more book items, while the responsibility of updating the button and remaining count could be delegated to a separate function or component. By separating these concerns, the code would be more modular and easier to maintain.

- **`handleSearchFormSubmit`:** This event handler function handles the form submit event for the search form. While it performs the task of filtering the books based on the search criteria, it also takes care of updating the DOM elements related to the filtered book list. This function could be improved by further separating the responsibilities. The filtering logic could be moved to a separate function or class, which returns the filtered result. Then, another function or component can handle updating the DOM elements based on the filtered result. This separation of concerns would make the code more modular and easier to test and modify.

- **`initializeList` and `updateListButton`:** These two functions handle the initial rendering of the book list and updating the "Show more" button's text and remaining count. While they perform their intended tasks, they directly manipulate the DOM elements. These functions could be improved by utilizing a more structured approach, such as using a templating engine or a component-based framework/library. Separating the rendering logic from the DOM manipulation would provide better abstraction and reusability, making the code maintainable and testable.

---

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

- **Single Responsibility Principle (SRP):** Each function or class should have a single responsibility.
    -

  - `handleListButtonClick`: Separate the responsibility of loading more book items from the responsibility of updating the button and remaining count. Create a separate function or class to handle updating the button and count based on the current state of the book list.

  - `handleSearchFormSubmit`: Separate the responsibility of filtering the books based on search criteria from the responsibility of updating the DOM elements. Move the filtering logic to a separate function or class, which returns the filtered result. Then, have another function or component handle updating the DOM elements based on the filtered result.

  - `initializeList` and `updateListButton`: Extract the DOM manipulation logic into separate functions or components that are responsible for rendering and updating the book list. Utilize a templating engine or a component-based framework/library to provide a more structured approach to rendering the book list and updating the button.

- **Open/Closed Principle (OCP)**: Code entities should be open for extension but closed for modification.

  Refactor the functions to be more modular and reusable by abstracting specific behaviors into separate functions or classes. By doing so, you can extend the functionality without modifying the existing code. For example, create a separate function or class to handle the rendering and updating of the book list, making it easier to modify or extend the rendering logic without impacting other parts of the code.

- **Dependency Inversion Principle (DIP**): High-level modules should not depend on low-level modules. Both should depend on abstractions.

  Implement abstractions and depend on them rather than concrete implementations. For example, create interfaces or abstract classes that define the contract for handling book list rendering and updating, and have the functions or classes depend on these abstractions. This allows for flexibility in switching implementations and promotes loose coupling between modules.
