DWA_07.4 Knowledge Check_DWA7

---

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

- The function of **‘BookElement’**

This abstraction is a good one because it promotes code reusability and maintainability. Instead of manually creating the HTML structure for each book element throughout the codebase, I can use this function to generate book elements by providing the necessary data. If there is a need to change the structure or styling of the book element, it can be done in one place within the function, and all instances of the book elements will reflect the changes.

- The function of **‘UpdateList’**

This abstraction is a good one because it follows the Single Responsibility Principle, which states that a function or module should have a single responsibility. In this case, the responsibility of the handleListButtonClick function is to handle the click event on a list button and perform the necessary actions associated with it.

The handleListButtonClick function abstracts away the details of how the page number is incremented, how the list items are updated, and how the list button is updated. Other parts of the codebase can simply call this function to perform these actions without needing to know the implementation details.

- The event handler functions

Each event handler function is responsible for handling a specific user interaction or event. This modular approach allows for better organization and maintainability of the code. It isolates the logic related to each specific action, making it easier to understand, test, and modify independently. This abstraction promotes code readability and reusability, as each event handler function focuses on a single task. It also allows for easier debugging and troubleshooting since the code is divided into smaller, manageable units of functionality.

---

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

- The function of **‘handleSearchFormSubmit’**

The handleSearchFormSubmit function is the worst abstraction due to its multiple responsibilities and mixing of presentation and logic. It violates the Single Responsibility Principle by handling form submission, filtering books, updating data, scrolling, and manipulating the DOM. The function tightly couples with global variables and directly accesses other functions. The nested loops and conditionals make the code hard to understand. Additionally, the search algorithm is inefficient for large datasets.

- **‘If’** statement of **‘active’** inside the function of **‘handleListItemsClick’**

The 'if(active)' part can be considered a poor abstraction because it tightly couples the code to the specific DOM structure and presentation. It directly accesses and manipulates the DOM elements using query selectors, which makes the code less modular and harder to test or reuse.

- the initialization of two variables: **‘page’** and **‘matches’**.

The code at first had a clear documentation and was maintainable, the changes were not necessary and that is why it is a worst abstraction as it would have been okay even if I didn’t change it into an object with a single property ‘number’ initialized to the value 1 and .

---

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

- Improving the **‘handleSearchFormSubmit’** function:

To improve the handleSearchFormSubmit function based on SOLID principles, I can apply the Single Responsibility Principle (SRP) by separating the responsibilities into smaller, more focused functions. For example, I can extract the filtering logic into a separate function responsible for filtering the books based on search criteria. This promotes better code organization and makes the code more maintainable and testable.

- Refactoring the if statement in **‘handleListItemsClick’**:

To improve the if statement in the handleListItemsClick function, I can apply the Open-Closed Principle (OCP) by utilizing polymorphism and abstraction. Instead of directly checking the value of active, I can define an abstraction, such as an interface or a base class, and implement different concrete classes based on different conditions. Each concrete class can encapsulate the specific behavior for a particular condition, promoting extensibility and reducing the need for modifying the if statement when new conditions arise.

- Improving the initialization of **‘page’** and **‘matches’**:

To improve the initialization of pages and matches, I can leverage the Dependency Inversion Principle (DIP) by injecting these dependencies into the respective functions or modules that require them. Rather than directly initializing the variables in the global scope, we can pass them as parameters to the functions that need them. This decouples the dependencies, improves testability, and allows for easier substitution of different implementations in the future.
