DWA_07.4 Knowledge Check_DWA7

---

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

1.1Modularization: The code is divided into multiple functions and separated by logical blocks, such as creating book previews, handling form submissions, applying filters, and event handling. This modular approach helps in organizing the code and promotes reusability and maintainability.

1.2. Separation of Concerns: The code separates different concerns, such as handling UI elements, applying filters, and manipulating the DOM. Each function seems to have a clear responsibility, which makes the code easier to understand and modify.

1.3. Data-driven Approach: The script imports data from an external module (data.js) and uses it to generate book previews, populate dropdowns, and apply filters. This data-driven approach allows for flexibility in managing and updating the content of the application.

---

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

2.1Lack of Clear Interfaces: The codebase doesn't define clear interfaces or contracts between different components or modules. This could make it harder to maintain and extend the codebase in the future. By defining clear interfaces, you can decouple components and promote better modularity.

2.2Limited Error Handling: The code doesn't include comprehensive error handling mechanisms. Error scenarios, such as failed API requests or unexpected user inputs, are not adequately handled. Adding proper error handling and validation can improve the robustness and user experience of the application.

2.3Limited Testability: The script doesn't incorporate explicit testing mechanisms or utilize a testing framework. Adding automated tests, such as unit tests or integration tests, can help identify and prevent regressions, improve code quality, and increase confidence in the application's behavior.

---

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

3.1Single Responsibility Principle (SRP): By adhering to the SRP, I can ensure that each class or module has a single responsibility. This can help improve the codebase's maintainability and make it easier to understand and modify.

3.2Interface Segregation Principle (ISP): The ISP states that clients should not be forced to depend on interfaces they do not use. In the context of the provided code, I can review the interfaces or contracts between different components or modules. This can help reduce dependencies and improve code maintainability.

3.3Dependency Inversion Principle (DIP): The DIP promotes depending on abstractions rather than concrete implementations. In my code,I can identify areas where modules or functions directly depend on concrete implementations, making them tightly coupled. Introduce abstractions, such as interfaces or abstract classes, to represent the dependencies and make them injectable into the dependent modules. This allows for easier substitution of implementations and improves the flexibility and testability of the code.
