DWA_07.4 Knowledge Check_DWA7

---

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

1: Functional Programming Abstraction:
Functional programming emphasizes the use of pure functions and immutable data, abstracting away mutable state and side effects. It enables developers to reason about code more easily, promotes code modularity, and supports parallel and distributed processing.
2: Database Abstraction:
Database abstraction layers provide a higher-level interface for interacting with databases, hiding the underlying complexities of different database systems. They offer a unified API to perform common database operations, making it easier to switch between different databases without modifying application code. This abstraction enhances flexibility, portability, and scalability of database interactions.
3: Event-driven Abstraction:
Event-driven programming abstracts the flow of a program by using events as a central communication mechanism. It allows components or modules to communicate through events, decoupling them and enabling asynchronous and loosely coupled systems.

---

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

1: Object-Oriented Programming (OOP) Abstraction:
OOP provides a powerful way to abstract real-world entities into classes, encapsulating data and behavior. It allows for modularity, reusability, and easier maintenance. By abstracting complex systems into classes and objects, OOP enables clear separation of concerns and promotes code organization.
2: Algorithmic Abstraction:
Algorithmic abstractions provide high-level algorithms or data structures that encapsulate complex computational tasks. They abstract away the low-level implementation details, allowing developers to use efficient and optimized algorithms without needing to understand their inner workings.
3: Dependency Injection (DI) Abstraction:
DI is a design pattern that abstracts the creation and management of dependencies from the components that use them. By injecting dependencies into components rather than having components create or manage their dependencies directly, DI enables better testability, decoupling, and flexibility in swapping out implementations.

---

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

Single Responsibility Principle (SRP):

- **Identify the distinct responsibilities within the bloated class or module.**
- **Refactor the code to create separate classes or modules, each with a single responsibility.**

- **Ensure that each class or module is focused on a specific task and has minimal dependencies on other components.**

Open/Closed Principle (OCP):

- **Identify areas of the code that are prone to frequent modifications or extensions.**
- **Design the code to be open for extension without modifying existing code.**
- **Use abstraction, interfaces, or abstract classes to define contracts that can be implemented or extended by different classes.**
- **Encapsulate varying behavior behind abstractions, allowing new functionality to be added without modifying existing code.**

Liskov Substitution Principle (LSP):

- **Ensure that subclasses can be substituted for their base classes without affecting the correctness of the program.**
- **Design class hierarchies with careful consideration of their behavior and contracts.**
- **Avoid violating expectations and preconditions when using polymorphism and inheritance.**
- **Refactor the code to ensure that subclasses adhere to the same contracts and behaviors as their base classes.**

Interface Segregation Principle (ISP):

- **Split large and monolithic interfaces into smaller, more focused interfaces.**
- **Clients should only depend on the interfaces they actually use.**
- **Identify the specific needs of clients and create interfaces tailored to those needs.**
- **This helps reduce unnecessary coupling and ensures that clients are not forced to depend on irrelevant or unnecessary methods.**

Dependency Inversion Principle (DIP):

- **Depend on abstractions rather than concrete implementations.**
- **Use interfaces or abstract classes to define dependencies.**
- **Inject dependencies into classes through constructor injection, method injection, or property injection.**
- **This promotes loose coupling, allows for interchangeable implementations, and facilitates easier testing and mocking of dependencies.**
