DWA_07.4 Knowledge Check_DWA7

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

- Object-Oriented Programming: It allows developers to represent real-world concepts as objects with their own attributes (data) and methods (behavior).
- Relational Databases: It provides an abstraction for storing and managing structured data into tables, this simplifies the management and retrieval of data, making it easier to model and work with complex data structures.
- Virtualization: It allows efficient utilization of hardware resources by running multiple virtual machines on a single physical server. This enables improved scalability, enhanced security through isolation, simplified management, and the ability to migrate or replicate virtual instances across different physical machines.

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

- Global State: It refers to the practice of storing and accessing data or variables in a globally accessible manner throughout a program. Debugging can become challenging because changes to the global state can have unintended consequences throughout the system.
- Over-Optimization: It refers to the practice of optimizing code prematurely without a clear understanding the execution and this often will result in a complex code which is hard to read
- Monolithic Architecture: It refers to an approach where an entire application or system is built as a single unit. Since all the functionality and components are packaged together, this hinders code readability, maintainability, and the ability to adapt to changing requirements.

---

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

Global State:

- SRP suggests that a module or class should have a single responsibility. By adhering to this principle, you can encapsulate and manage state within individual modules, limiting the exposure of global state. This promotes better organization, readability, and maintainability.
- DIP promotes dependency inversion and encourages the use of abstractions. By depending on abstractions rather than concrete implementations, you can minimize direct dependencies on global state. Components can interact through well-defined interfaces, making it easier to modify the underlying state management without affecting the entire system.

Monolithic Architecture:

- The Single Responsibility Principle and the Interface Segregation Principle advocate for creating smaller, cohesive modules that have clear responsibilities and well-defined interfaces. Breaking down a monolithic system into smaller, loosely-coupled modules aligns with these principles and promotes modularity and maintainability.
- The Dependency Inversion Principle can be applied by designing modules to depend on abstractions rather than concrete implementations. This allows modules to be easily modified without affecting the rest of the system. Adopting a dependency injection framework can aid in applying the DIP effectively.

Over-Optimization:

- The Single Responsibility Principle encourages writing classes or modules that have a single responsibility. By keeping code focused on its primary responsibility, it becomes easier to understand, modify, and optimize when necessary.
- The Open/Closed Principle promotes the use of abstraction and extension rather than modification. This principle allows you to design code that is open to extension but closed to modification. By adhering to this principle, you can focus on building a flexible and adaptable codebase, which can aid in making targeted optimizations when needed.