DWA_07.4 Knowledge Check_DWA7

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

Object-Oriented Programming (OOP):
Object-Oriented Programming is a programming paradigm that focuses on the concept of objects, which are instances of classes that encapsulate data and behavior. OOP provides a way to model real-world entities and relationships, making it easier to manage and organize complex codebases. The key benefits of OOP include encapsulation, inheritance, and polymorphism, which enable code reuse, modularity, and extensibility. OOP has been widely adopted and supported by many programming languages like Java, C++, and Python.

Virtual Machines (VMs):
A virtual machine is an abstraction layer that emulates a complete computer system, allowing multiple operating systems or software applications to run on a single physical machine. VMs provide isolation, portability, and resource management, enabling more efficient use of hardware resources. They have revolutionized software development and deployment by enabling the creation of virtualized environments that can run different software configurations without interference. Popular VM technologies include VMware, VirtualBox, and Hyper-V.

Relational Databases:
Relational databases provide an abstraction for storing and organizing data in tables, consisting of rows and columns. They are based on the relational model and use Structured Query Language (SQL) for querying and manipulating data. The key advantages of relational databases include data integrity, scalability, and the ability to define relationships between different tables. They have been widely used in various applications and industries, providing reliable and efficient data storage and retrieval. Popular relational database management systems (RDBMS) include MySQL, Oracle Database, and Microsoft SQL Serv

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

Global Variables:
Global variables are variables that can be accessed and modified from any part of a program. While they may seem convenient, excessive use of global variables can lead to various issues. Global variables can introduce tight coupling, making code more difficult to understand, test, and maintain. They can also make it challenging to identify the source of bugs and can cause unexpected side effects when modified from multiple locations. Therefore, it's generally recommended to limit the use of global variables and favor more localized data sharing mechanisms like function parameters or object properties.

Monolithic Architecture:
Monolithic architecture refers to a software design approach where an application is built as a single, indivisible unit. In this architecture, all components and functionalities are tightly coupled, making it difficult to scale, maintain, and modify the system. Changes to one part of the application often require retesting and redeployment of the entire system. Monolithic architectures can be challenging to parallelize and can suffer from performance bottlenecks. To address these issues, modular and microservices architectures have gained popularity as they provide better scalability, maintainability, and flexibility.

Spaghetti Code:
Spaghetti code is a term used to describe poorly structured or unorganized code that is difficult to understand and maintain. It typically arises when a program lacks clear structure, contains tangled control flow, and has excessive dependencies between different parts of the codebase. Spaghetti code can be problematic because it is challenging to debug, modify, and extend. It lacks separation of concerns, making it harder to reason about the behavior of the system. Applying software engineering principles like modularization, encapsulation, and abstraction can help alleviate spaghetti code issues.

---

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

Global Variables:
To mitigate the problems associated with global variables, the Single Responsibility Principle (SRP) and the Dependency Inversion Principle (DIP) from the SOLID principles can be helpful.
SRP suggests that a class or module should have a single responsibility. By applying SRP, you can encapsulate the shared state within a class or module, reducing the need for global variables. This allows for better organization, separation of concerns, and easier maintenance.

DIP recommends that high-level modules should not depend on low-level modules directly, but both should depend on abstractions. By applying DIP, you can define interfaces or abstractions that represent the shared state or functionality instead of relying directly on global variables. This enables better decoupling, testability, and flexibility.

Monolithic Architecture:
To address the limitations of a monolithic architecture, several SOLID principles can be beneficial, including the Single Responsibility Principle (SRP), the Open/Closed Principle (OCP), and the Dependency Inversion Principle (DIP).
SRP suggests that a class or module should have a single responsibility. By following SRP, you can identify cohesive units of functionality within a monolithic application and extract them into separate modules or microservices. This allows for better scalability, maintainability, and separation of concerns.

OCP promotes the idea that software entities (classes, modules, etc.) should be open for extension but closed for modification. By designing your monolithic application with OCP in mind, you can ensure that changes and new functionalities can be added without modifying the existing code. This can be achieved through the use of interfaces, abstractions, and dependency injection.

DIP suggests that high-level modules should not depend on low-level modules directly, but both should depend on abstractions. By applying DIP, you can modularize your monolithic application into smaller, loosely coupled components that depend on abstractions. This allows for easier testing, swapping of implementations, and overall flexibility.

Spaghetti Code:
To improve spaghetti code, several SOLID principles can be helpful, including the Single Responsibility Principle (SRP), the Open/Closed Principle (OCP), and the Dependency Inversion Principle (DIP).
SRP encourages breaking down complex code into smaller, focused modules that have a single responsibility. By following SRP, you can create more cohesive and maintainable code, making it easier to understand, modify, and test.

OCP promotes designing code that is open for extension but closed for modification. By adhering to OCP, you can avoid making widespread changes to existing code when adding new features or modifying behavior. This can be achieved by leveraging interfaces, abstract classes, and polymorphism.

DIP suggests depending on abstractions instead of concrete implementations. By applying DIP, you can decouple modules, reduce dependencies, and improve testability. By relying on abstractions, you can easily swap out implementations and enforce loose coupling between different parts of the codebase.