DWA_07.4 Knowledge Check_DWA7

---

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

Interfaces or Protocols: These are used to define a set of methods that must be implemented by a class2. This allows for the creation of multiple classes that implement the same interface, allowing for code reuse and flexibility.
Abstract Classes: These are classes that cannot be instantiated and are used to provide a common interface for subclasses2. They can contain both abstract methods (methods without an implementation) and concrete methods (methods with an implementation).
Function Types/References/Signatures: These are used to define the type of a function, including its input and output. This allows for the creation of functions that can be passed as arguments to other functions, allowing for code reuse and flexibility.

---

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

Over-abstraction: This occurs when code is abstracted to the point where it becomes difficult to understand and maintain1. It can result in code that is overly complex and difficult to work with.
Leaky abstractions: This refers to abstractions that do not completely hide the underlying implementation details2. As a result, users of the abstraction may need to have knowledge of the underlying implementation in order to use it effectively.
The wrong abstraction: This occurs when an abstraction is created that does not accurately represent the underlying concept or object1. It can result in code that is difficult to understand and maintain, and may require significant refactoring to fix.

---

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

Single Responsibility Principle: This principle states that a class should have one and only one reason to change, meaning that a class should have only one job1. By adhering to this principle, over-abstraction can be avoided as each class will have a specific and well-defined responsibility.
Open-Closed Principle: This principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification1. By adhering to this principle, leaky abstractions can be avoided as the underlying implementation details will be hidden and the abstraction will be properly encapsulated.
Liskov Substitution Principle: This principle states that objects of a superclass should be able to be replaced with objects of a subclass without altering the correctness of the program1. By adhering to this principle, the wrong abstraction can be avoided as the abstraction will accurately represent the underlying concept or object.
