DWA_07.4 Knowledge Check_DWA7

---

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

1.1 Modules/ theme.js

- By separating the code into modules, it becomes easier to manage and work with.
- Another advantage is that we can reuse code from one module in another module.
- It's like having a set of building blocks that we can use in different projects without having to rewrite them every time.
- This makes it easier to collaborate with others on a project since everyone can work on different modules independently.
- It also helps with debugging and testing since we can isolate and test each module separately.
- Overall, by creating separate module files and exporting their functionality, we can have organized, reusable, and easier-to-maintain code.

1.2 Breaking code into smaller pieces

- By breaking the code into smaller pieces, you isolate specific functionality, improve the clarity and organization of the code, and enable better code reuse and maintainability.
- Smaller code segments are easier to test.

1.3 Using Functions

- This code can be reused throughout your program, You can call the same function multiple times with different inputs, saving time and effort.

- Functions can accept parameters, which allows you to make them flexible and adaptable. By passing different values as arguments, you can customize the behavior of a function. This parameterization makes your code more versatile, as you can reuse the same function with different inputs.
- promote code maintainability by encapsulating specific functionality. If a bug is found or a change is needed, you only need to modify the function in question, rather than searching through the entire codebase. This makes it easier to update, fix, and enhance your code over time.

---

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

1.1 Classes

- Making changes to one part of a class-based system can affect other parts.
- Understanding classes and related concepts may take time to learn. It's like learning a new language or understanding how different pieces of a puzzle fit together.
- Using classes can sometimes make programs slower. It's like adding extra steps to a recipe that can take longer to finish baking the cake.
- Working with classes can sometimes be complicated, especially if the code is very large or has many interconnections. It's like having a puzzle with lots of pieces that need to fit together just right.

1.2 Using Constants

- When multiple constants are defined throughout the codebase, there is a risk of name collisions or conflicts.

1.3 Higher-order functions

- Using higher-order functions can introduce additional complexity when debugging code. With functions passed as arguments or returned as results, it may be challenging to track the flow of execution and identify issues in the code.

---

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

1.1 Classes

Single Responsibility Principle (SRP): This principle states that a class should have only one reason to change. By ensuring that each class has a single responsibility, it becomes easier to understand, test, and maintain.

1.2 Constants

The Single Responsibility Principle (SRP) means that a class or module should only have one main job. When using constants, it's important to place them in modules or classes that are focused on a specific purpose related to those constants. By organizing constants based on their purpose or domain, it becomes easier to manage and modify them when necessary.

Instead of spreading constants all over the code, it's better to create separate modules or classes for different areas or domains that need constants. This way, each module or class will be responsible for a specific group of related constants, making them easier to handle and maintain.

1.3 Higher-order functions

Liskov Substitution Principle (LSP): Higher-order functions should adhere to the LSP by ensuring that functions passed as arguments comply with the expected contract or

interface. This allows for interchangeable function implementations without affecting the behavior of the higher-order function. By following LSP, you can easily substitute

---