DWA_01.3 Knowledge Check_DWA1

---

1. Why is it important to manage complexity in Software?

When we build software, it naturally becomes **complex due to various factors like requirements, interactions, and system size**. This complexity is unavoidable.

Managing Complexity: Instead of trying to eliminate complexity entirely (which is impossible), we focus on managing it. Here's how:

Redistributing Complexity: Different solutions shift complexity around. For example, microservices **distribute complexity across smaller parts of an application**.

---

2. What are the factors that create complexity in Software?

Accidental Complexity: accidental complexity is counterproductive. **It results from poor understanding, communication, and practices.** Examples include bad architecture, design, and code. The solution? Education, training, and continuous improvement to minimize accidental complexity.

Incidental Complexity: This is the trickiest type. **It involves dealing with tools and technologies that aren't directly related to the problem we're solving**. For instance, load balancers, databases, and containers.

---

3. What are ways in which complexity can be managed in JavaScript?

- Asynchronous Programming:

JavaScript's asynchronous model allows tasks to run concurrently, preventing blocking.

Key concepts include:

Callbacks: Use callback functions to handle asynchronous operations.

Promises: Promises provide a cleaner way to manage asynchronous code.

async/await: Modern syntax for handling asynchronous operations.

- Functional Programming:

Functional programming treats computation as the evaluation of mathematical functions.

Leverage first-class functions and higher-order functions for cleaner code.

- Prototypal Inheritance:

JavaScript uses prototypal inheritance, where objects inherit directly from other objects via the prototype chain.

Understand how objects inherit properties and methods.

- Memory Management and Garbage Collection:

Be aware of memory usage and prevent memory leaks.

JavaScript has automatic garbage collection, deallocating objects no longer referenced.

- Performance Optimization Techniques:

Optimize code by minimizing DOM operations, using efficient loops, and avoiding unnecessary function calls.

Choose efficient algorithms for better performance.

---

4. Are there implications of not managing complexity on a small scale?

- Readability and Maintainability:

Complex code is harder to read and understand. When you revisit your code later, you might struggle to remember its purpose or how it works.

Poorly managed complexity leads to “spaghetti code,” making maintenance and debugging challenging.

- Bugs and Errors:

Complex code tends to hide subtle bugs. When different parts of your code interact in intricate ways, unexpected issues can arise.

Unmanaged complexity increases the likelihood of logic errors, null pointer exceptions, and other runtime issues.

- Scalability:

Small-scale projects often grow over time. If you don't manage complexity early, it can hinder future enhancements.

Adding new features becomes difficult when the existing codebase lacks clear structure.

- Collaboration:

If you work in a team, unmanaged complexity affects collaboration. Team members may struggle to understand each other's code.

Clear code organization and consistent patterns facilitate collaboration.

- Performance:

Complex code can be inefficient. Redundant calculations, excessive loops, or unnecessary function calls impact performance.

Optimizing complex code becomes harder as it accumulates technical debt.

- Testing and Debugging:

Testing complex code requires more effort. You need to cover various execution paths and edge cases.

Debugging becomes a puzzle, especially if you're dealing with intricate interactions.

- Time and Productivity:

Unmanaged complexity slows down development. You spend more time deciphering code than implementing features.

Simple, well-organized code allows you to focus on solving the problem at hand.

---

5. List a couple of codified style guide rules, and explain them in detail.

- Choosing a Format:

  Opinions on indentation, whitespace, and line lengths can be controversial. To maintain consistency, use a code formatter like **Prettier**. It ensures consistent style and avoids off-topic discussions.

  Additionally, follow these rules:

➔ Use **literals** (e.g., `const visitedCities = []`) instead of constructors when creating arrays.
➔ Add items to an array using `push()` (e.g., `pets.push("cat")`) rather than direct assignment (`pets[pets.length] = "cat"`).
➔ Write **asynchronous code** using Promises or `async/await` for better performance.

- **Comments**:

➔ Comments are crucial for clarity. They help developers understand code intent.
➔ Add comments when the purpose or logic isn't obvious.

- **Variable Naming**:

➔ Choose descriptive variable names that convey their purpose. Avoid single-letter variable names (unless they represent common conventions like `i`, `j`, or k in loops).
➔ Use camelCase for variables and functions (e.g., `myVariable`, `calculateTotal`).
➔ For constants, use uppercase with underscores (e.g., `MAX_LENGTH`, `PI`).

- **Avoid Global Variable**s:

➔ Minimize the use of global variables. They can lead to unintended side effects and make code harder to reason about.
➔ Instead, use local scope (inside functions or blocks) and pass data explicitly.

---

6. To date, what bug has taken you the longest to fix - why did it take so long?

Using the correct syntax.
