### **Adapter Pattern Summary**

1. **Purpose**:
   * Provides a way to adapt or bridge incompatible interfaces so they can work together.
   * Typically used to integrate new code with legacy code or external APIs that cannot be modified.
2. **Ease of Implementation**:
   * Simple and quick to implement.
   * Offers a straightforward solution to a specific compatibility problem.
3. **Common Use Cases**:
   * Primarily used to allow new systems to interface with legacy code.
   * Useful for creating adapters for external APIs without altering the original codebase.
4. **Best Practices**:
   * Don’t stop at a single adapter; consider using multiple adapters to handle various integration points within the application.
   * Evaluate additional opportunities to use adapters for other incompatible systems or APIs.
5. **Key Advice**:
   * Use the Adapter Pattern whenever you need to make incompatible interfaces work together, especially with legacy systems.
   * Be open to creating multiple adapters to maximize integration flexibility.

### **Composite vs. Decorator Pattern Summary**

1. **Composite Pattern**:
   * **Structure**: Creates a **tree structure** where both individual objects (leaves) and groups of objects (composites) share the same interface.
   * **Purpose**: Unifies object structures, allowing clients to treat individual and composite objects uniformly.
   * **Use Case**: Ideal when you want to represent part-whole hierarchies, such as folder and file structures.
2. **Decorator Pattern**:
   * **Structure**: Uses **composition** to add functionality by "wrapping" an object with additional behavior.
   * **Purpose**: Modifies or extends the behavior of a specific object without changing its class.
   * **Use Case**: Suitable when you need to add new behaviors or responsibilities to objects dynamically without altering their underlying structure.
3. **Key Differences**:
   * **Composite**: Organizes objects in a hierarchy, focusing on representing groups and individuals with the same interface.
   * **Decorator**: Focuses on modifying behavior of an object by containing (wrapping) it, allowing additional functionality to be layered on without altering the base object.
4. **Best Practices**:
   * Use **Composite** for creating and managing object hierarchies with unified interfaces.
   * Use **Decorator** to add or override behaviors in objects without subclassing, especially for dynamic modifications.
5. **Key Advice**:
   * Choose Composite if you need to work with tree-like structures.
   * Choose Decorator if you want to enhance functionality in specific objects without changing their core class.

### **Façade Pattern Summary**

1. **Purpose**:
   * Simplifies the interface for the client by hiding complex or "ugly" parts of an API.
   * Reduces dependencies and makes the system easier to use by providing a clean, high-level interface.
2. **Ease of Implementation**:
   * Very straightforward to implement.
   * Often used as a **refactoring tool** to clean up and simplify existing code.
3. **When to Use**:
   * Ideal for complex systems with multiple classes or dependencies that clients don’t need to interact with directly.
   * Useful for reducing the amount of information the client needs to know, enhancing maintainability and readability.
4. **Common Pitfalls**:
   * Like the Singleton Pattern, it’s easy to overuse because it seems helpful and simple to implement.
   * Be careful not to add unnecessary façade layers; use it only when it genuinely simplifies interactions.
5. **Best Practices**:
   * Apply the Façade Pattern as a **post-design refactoring** tool rather than designing it upfront.
   * Use it when you realize the client is managing too many dependencies or dealing with too much internal complexity.
6. **Example Use Case**:
   * In scenarios like a database client (e.g., GDBC), without the façade, the client would need to handle connections, statements, and result sets directly.
   * With a façade, the client only interacts with a single interface, while the façade manages all the underlying details like exception handling and connections.
7. **Key Advice**:
   * Use the Façade Pattern to encapsulate complex subsystems and reduce dependencies, making the codebase cleaner and more maintainable.
   * Add façades as needed to refactor complex code, not as an initial design feature.

### **Proxy vs. Decorator Pattern Summary**

1. **Proxy Pattern**:
   * **Purpose**: Primarily controls access to an object or adds functionality indirectly, such as handling remote, virtual, or protected access to a resource.
   * **Functionality**: Although it can add functionality, its main role is not to enhance behavior but to mediate and manage access.
   * **Timing**: The proxy's behavior is determined at **compile-time**, meaning the exact class it intercepts is known and set upfront.
   * **Single Instance**: Typically, only one proxy is used per instance, as it directly represents or controls access to that instance.
2. **Decorator Pattern**:
   * **Purpose**: Designed explicitly to **dynamically add or extend functionality** to objects.
   * **Chaining**: Supports **chaining** multiple decorators together to layer or stack additional behaviors.
   * **Timing**: Decorator behavior is determined at **runtime**, allowing flexibility to add or remove decorations dynamically.
   * **Self-Referencing**: Decorators can reference others of the same type within their chain, forming a stack-like structure.
3. **Key Differences**:
   * **Proxy**: Focuses on access control or managing resource access (e.g., remote calls), with fixed functionality at compile-time.
   * **Decorator**: Focuses on enhancing and adding features, allowing behaviors to be layered at runtime.
4. **Use Cases**:
   * Use **Proxy** when you need controlled access to an object, such as for lazy loading, remote proxies, or access control.
   * Use **Decorator** when you want to add or modify functionality dynamically without altering the object's core class.
5. **Key Advice**:
   * Choose Proxy for access management, particularly where only one level of control is required.
   * Choose Decorator for flexible, stackable functionality changes that can be applied or removed as needed at runtime.