**Conceptual Questions**

1. **What is the Facade Design Pattern?**
   * Answer: The Facade pattern provides a simplified, unified interface to a set of interfaces in a subsystem. It hides the complexities of the system and provides an easy-to-use interface for the client.
2. **How is the Facade pattern different from the Adapter pattern?**
   * Answer: The Facade pattern provides a higher-level interface to simplify interaction with a subsystem. The Adapter pattern, on the other hand, is used to make two incompatible interfaces work together.
3. **What are the advantages of using the Facade pattern?**
   * Simplifies the client interface.
   * Reduces dependency between clients and subsystems.
   * Improves code readability and maintainability.
4. **When would you choose to use the Facade pattern?**
   * Use it when:
     + You want to simplify access to a complex system.
     + You want to decouple clients from subsystem classes.
     + You need to provide a unified interface to a group of related classes.
5. **What are the potential drawbacks of the Facade pattern?**
   * Overuse of facades can lead to god objects that handle too many responsibilities.
   * It can hide too much of the system, limiting flexibility for advanced users.

**Scenario-Based Questions**

1. **Can you describe a real-world example where you would use the Facade pattern?**
   * Example: In an online shopping system, you can create a OrderService facade that interacts with subsystems like PaymentService, ShippingService, and InventoryService. The facade provides a single method like placeOrder() that handles all these internal operations.
2. **How would you implement a Facade pattern for a home automation system?**
   * Answer: Create a HomeAutomationFacade class that interacts with subsystems like LightSystem, ClimateControl, and SecuritySystem. The facade provides methods like activateMorningMode() or activateNightMode() to control multiple subsystems.
3. **Suppose a subsystem changes frequently. How does the Facade pattern help in such scenarios?**
   * Answer: The Facade pattern shields the client from direct interaction with the subsystem. Even if the subsystem changes, the facade's interface can remain the same, ensuring that the client code doesn't need modification.

**Code-Related Questions**

1. **Write code to demonstrate the Facade pattern for a Movie Streaming Service.**
   * Example:

python

Copy code

class VideoStreaming:

def load\_video(self, video\_id):

return f"Loading video {video\_id}"

class AudioStreaming:

def load\_audio(self, video\_id):

return f"Loading audio for video {video\_id}"

class Subtitles:

def load\_subtitles(self, video\_id):

return f"Loading subtitles for video {video\_id}"

class StreamingFacade:

def \_\_init\_\_(self):

self.video = VideoStreaming()

self.audio = AudioStreaming()

self.subtitles = Subtitles()

def play\_movie(self, video\_id):

print(self.video.load\_video(video\_id))

print(self.audio.load\_audio(video\_id))

print(self.subtitles.load\_subtitles(video\_id))

# Client code

facade = StreamingFacade()

facade.play\_movie("123")

1. **How would you refactor tightly coupled code using the Facade pattern?**
   * Answer: Identify the parts of the code interacting with multiple subsystems and introduce a facade class that encapsulates these interactions. Replace direct calls to subsystems with calls to the facade class.

**Design and Architecture Questions**

1. **How does the Facade pattern improve the scalability of a system?**
   * Answer: By providing a single entry point to a complex subsystem, the Facade pattern reduces the number of dependencies. This makes it easier to scale the system, as changes in subsystems don't affect clients.
2. **Can you use the Facade pattern with other design patterns?**
   * Answer: Yes. Common combinations include:
     + **With Singleton**: To ensure a single facade instance.
     + **With Factory**: To create subsystem objects behind the facade.
     + **With Adapter**: To adapt subsystem interfaces before exposing them through the facade.
3. **How does the Facade pattern adhere to the principle of loose coupling?**
   * Answer: By decoupling the client from the subsystem. The client only interacts with the facade, not the individual subsystem components.
4. **What would you do if a subsystem needs to expose additional functionality to some clients but not others?**
   * Answer: Create multiple facades or extend the existing facade with specific methods for advanced clients, while keeping the basic methods for simpler use cases.

**Advanced Questions**

1. **What would happen if a Facade class becomes too large? How would you address this issue?**
   * Answer: A large Facade class indicates it might be taking on too many responsibilities. To resolve this:
     + Split the facade into multiple smaller facades, each responsible for a specific subsystem.
     + Use the Composite pattern to manage multiple facades under a single entry point.
2. **How does the Facade pattern affect testing?**
   * Answer: The Facade pattern makes testing easier by providing a unified interface. Tests can focus on the facade's behavior rather than testing each subsystem component individually.
3. **What is the difference between a Facade and an API Gateway?**
   * Answer:
     + **Facade**: Simplifies access to a subsystem within a single application.
     + **API Gateway**: Centralizes access to multiple microservices in a distributed system, often handling cross-cutting concerns like authentication and logging.
4. **How can you design a facade for a microservice-based architecture?**
   * Answer: Create a centralized service (facade) that orchestrates calls to various microservices and exposes a simplified API for clients. For example, an OrderServiceFacade can handle requests for ordering, shipping, and billing by interacting with their respective microservices.

**Debugging and Real-Life Issues**

1. **What would you do if a subsystem exposes private functionality that the Facade needs to access?**
   * Answer: Refactor the subsystem to expose only the required methods through a public interface or use dependency injection to grant the facade access to the subsystem's private methods.
2. **If a client bypasses the facade and accesses subsystems directly, how would you prevent this?**
   * Answer: Restrict access to subsystem classes by:
     + Making subsystem classes package-private (in languages like Java).
     + Using access modifiers to limit visibility.
     + Configuring proper architectural boundaries in the codebase.