Interview Questions and Answers on SOLID Design Principles

1. **Can you name the five SOLID principles?**
   The SOLID design principles are a set of guidelines that aim to help developers create more robust and maintainable software. **Robert C. Martin** introduced these 5 principles in his 2000 paper "**Design Principles and Design Patterns**". **The acronym** stands for
   S - Single Responsibility Principle
   O - Open Closed Principle
   L - Liskov Substitution Principle
   I - Interface Segregation Principle
   D - Dependency Inversion Principle

2. **Can you explain briefly the SOLID design principles?**
   1. **Single Responsibility Principle (SRP):** A class should have only one reason to change, meaning it should have only one responsibility.
   2. **Open/Closed Principle (OCP):** Software entities (classes, modules, etc.) should be open for extension but closed for modification.
   3. **Liskov Substitution Principle (LSP):** Objects of a superclass should be replaceable with objects of its subclasses without breaking the correctness of the program.
   4. **Interface Segregation Principle (ISP):** A client should not be forced to depend on methods it does not use. Instead, it should depend on interfaces that are specific to its needs.
   5. **Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules. Instead, they should both depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

3. **How do you ensure Single Responsibility Principle (SRP) in your code?**
   To ensure Single Responsibility Principle (SRP) in my code, I follow these guidelines:
   - Each class should have only one responsibility.
   - The class should have only one reason to change.
   - The class should do one thing, and do it well.

4. **How do you implement Open-Closed Principle (OCP) in your code?**
   To implement the Open-Closed Principle (OCP) in my code, I follow these guidelines:
   - Design classes to be easily extended.
   - Use abstractions such as abstract classes or interfaces to define extension points.
   - Avoid making changes to existing code when adding new functionality.

5. **How do you ensure Liskov Substitution Principle (LSP) in your code?**
   To ensure Liskov Substitution Principle (LSP) in my code, I follow these guidelines:
   - Ensure that subtypes can be substituted for their base types without affecting the correctness of the program.
   - Subtypes should not change the preconditions or post-conditions of the methods of their base types.
   - Subtypes should not introduce new exceptions that are not part of the exception hierarchy of their base types.

6. **How do you apply Interface Segregation Principle (ISP) in your code?**
   To apply Interface Segregation Principle (ISP) in my code, I follow these guidelines:
   - Define small, focused interfaces that are specific to the needs of the clients that use them.
   - Avoid creating large, monolithic interfaces that force clients to implement methods they don't need.
   - Prefer multiple, specialized interfaces over a single, general-purpose interface.

7. **How do you apply Dependency Inversion Principle (DIP) in your code?**
   To apply Dependency Inversion Principle (DIP) in my code, I follow these guidelines:
   - Define high-level modules that depend on abstractions, not on concrete implementations.
   - Define low-level modules that provide concrete implementations for the abstractions used by high-level modules.
   - Use dependency injection to provide the concrete implementations to the high-level modules.

8. **How do SOLID principles help in achieving maintainable code?**

   SOLID principles help achieve maintainable code by promoting code that is easier to understand, modify, and extend. By following SOLID principles, code becomes more modular, cohesive, and loosely coupled, making it easier to test and maintain. For example, following the **Single Responsibility Principle** (SRP) ensures that each class and function has a single responsibility, reducing the complexity of the code. Similarly, adhering to the **Open-Closed Principle** (OCP) allows for code to be extended without modifying existing code, reducing the risk of introducing bugs. Overall, SOLID principles lead to code that is more maintainable and adaptable to changing requirements.

9. **What are the common violations of SOLID principles?**

   Common violations of SOLID principles include:

   - Violating the **Single Responsibility Principle** (SRP) by having a class or function that does too many things.
   - Violating the **Open-Closed Principle** (OCP) by modifying existing code rather than extending it through new code.
   - Violating the **Liskov Substitution Principle** (LSP) by violating the contract defined by the base class.
   - Violating the **Interface Segregation Principle** (ISP) by forcing clients to depend on interfaces they don't use.
   - Violating the **Dependency Inversion Principle** (DIP) by depending on concrete implementations rather than abstractions.

10. **What are the advantages of following SOLID principles in software development?**

    Following SOLID principles in software development has several advantages, including:

    - **Maintainability**: Code that adheres to SOLID principles is easier to maintain and modify over time, as it is structured in a clear and organized way.
    - **Flexibility**: SOLID code is also more flexible, as changes to one part of the system are less likely to have unintended consequences in other parts of the codebase.
    - **Testability**: SOLID code is easier to test, as it is designed with the principles of separation of concerns and dependency inversion in mind.

11. How Design Principles are different from Design Patterns?

Design Principles are the core principles which we are supposed to follow while designing any software system on any platform using any programming language.

Examples of Design Principles:

- Dependency on abstraction not concrete classes
- Encapsulate that varies
- Program to interfaces not implementations

Design Patterns are the solutions to common occurring general problems, they are not exact program that can be fit to solve your problem rather you need to customize according to your problem. Design Patterns are already invented solutions which are well tested and safe to use.

Examples of Design Patterns:

- Single Design Pattern: When you want only one instance of a class.
- Repository Design Pattern: To separate different layers of application (Business Repository, Data Repository)

12. What do you understand about coupling and cohesion in software development?

Coupling is the degree to which one software component is dependent on another. The more coupling there is between two components, the more difficult it is to make changes to one without affecting the other. Cohesion, on the other hand, is the degree to which a software component is self-contained. A component with high cohesion is one that is focused on a single task and is not tightly coupled to other components.

13. How do you reduce coupling and increase cohesion in your code?

One way to reduce coupling is to use dependency injection, which allows you to inject dependencies into a class rather than having the class create them itself. This way, the class is not tied to a specific implementation of the dependency. To increase cohesion, you can make sure that each class has a single responsibility, and that its methods are all related to that responsibility.

14. What are some best practices for avoiding duplicate code?

One best practice for avoiding duplicate code is to create abstract base classes that define the common functionality shared by the child classes. Another best practice

is to use interfaces to define the contracts that must be implemented by the child classes. Finally, it is also helpful to use design patterns to promote code reuse.

15. Why is tight coupling bad?

Tight coupling is when two classes are too closely linked together, and changes to one class can unintentionally break the other class. This can lead to code that is difficult to maintain and debug.

16. What are some ways to avoid using multiple inheritance in programming?

There are a few ways to avoid using multiple inheritance in programming. One way is to use interfaces instead of classes to define the functionality that a class should have. Another way is to use composition instead of inheritance, so that a class contains other classes instead of inheriting from them. Finally, you can use delegation, where one class delegates responsibility for certain tasks to another class.

17. Do you prefer loose or tight coupling in your programs? Why?

I prefer loose coupling in my programs because it allows for more flexibility and easier maintenance. If two components are tightly coupled, then a change in one component can potentially break the other component. With loose coupling, the components are more independent and can be changed without affecting the other components.

18. What is the difference between object oriented programming and procedural programming?

Object oriented programming is a programming paradigm that is based on the concept of objects. These objects are self-contained and can interact with each other. Procedural programming, on the other hand, is a programming paradigm that is based on the concept of procedures. These procedures are a series of steps that are followed in order to complete a task.

19. What are some cases where high cohesion can be seen as negative?

High cohesion can be seen as negative when it leads to code that is tightly coupled and difficult to change. This can happen when a class or module is trying to do too many things, and as a result, the code is difficult to understand and maintain.

20. What is OCP (Open/Closed Principle) and why is it important?

OCP is the Open/Closed Principle, which states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that new functionality can be added to an existing software entity without having to modify the existing code. This is important because it helps to prevent code breakage and makes code more maintainable.

21. What are the benefits of loose coupling in software development?

Loose coupling in software development has several benefits, including:

- **Flexibility**: loosely coupled code can be easily modified or extended without affecting other parts of the system, making it easier to adapt to changing requirements.
- **Testability**: loosely coupled code is easier to test because it can be tested in isolation, reducing the risk of bugs and improving the overall quality of the code.
- **Maintainability**: loosely coupled code is easier to maintain because changes can be made to a specific module without affecting the rest of the system.

22. How do you deal with tight coupling in your code?

To deal with tight coupling in code, one approach is to use design patterns and principles such as Dependency Inversion Principle (DIP) and Dependency Injection. By using DIP, high-level modules depend on abstractions instead of concrete implementations, reducing coupling. In addition, Dependency Injection allows dependencies to be injected into a class instead of being tightly coupled within the class itself. Here's an example of how Dependency Injection can be used to reduce tight coupling in a class.

23. What are some disadvantages of tightly coupled classes?

Tightly coupled classes can be difficult to maintain and test because they are so interdependent. If you need to make a change to one class, it can have a ripple effect on other classes, and this can make it difficult to predict how the system will behave. Additionally, tightly coupled classes can make it difficult to reuse code because you often need to use the entire system of classes rather than just one or two classes.

24. How would you identify if a codebase violates the Single Responsibility Principle (SRP)?

To identify if a codebase violates the Single Responsibility Principle (SRP), we can look for classes or methods that have more than one reason to change. If a class or method is responsible for more than one thing, it violates the SRP. Here's an example of a class that violates the SRP.

25. How would you refactor a codebase that violates the Open-Closed Principle (OCP)?

To refactor a codebase that violates the Open-Closed Principle (OCP), we can use a combination of inheritance and interfaces.

26. How would you ensure adherence to the Liskov Substitution Principle (LSP) when designing a new class hierarchy?

To ensure adherence to the Liskov Substitution Principle (LSP) when designing a new class hierarchy, we need to ensure that any subclass can be used in place of its parent class without affecting the correctness of the program. Here are some guidelines:

- Each subclass should implement all the methods of its parent class.
- The preconditions of the overridden methods in the subclass should not be stronger than those in the parent class.
- The postconditions of the overridden methods in the subclass should not be weaker than those in the parent class.
- The invariants of the parent class should be preserved in the subclass.

27. How would you refactor a codebase to adhere to the Interface Segregation Principle (ISP)?

To adhere to the Interface Segregation Principle (ISP), we should break down large interfaces into smaller and more specialized ones so that clients only need to depend on the interfaces that they actually use. Here are some steps we can take to refactor a codebase to adhere to ISP:

- Identify interfaces that are too large or that have methods that are not being used by their clients.
- Break down those interfaces into smaller and more specialized ones, based on the needs of the clients that use them.
- Make sure that clients only depend on the interfaces that they actually use.

28. How would you design a dependency injection container to adhere to the Dependency Inversion Principle (DIP)?

A dependency injection container is a tool used to implement the Dependency Inversion Principle (DIP) by providing a mechanism for loosely coupling objects and their dependencies. To design a dependency injection container that adheres to the DIP, you can define interfaces for your dependencies and then use those interfaces to inject concrete implementations.

29. How do you apply SOLID principles to achieve scalability in a distributed system?

To achieve scalability in a distributed system, we can apply SOLID principles as follows:

1. **Single Responsibility Principle (SRP):** Divide functionality into small, independent services, each with a single responsibility.
2. **Open-Closed Principle (OCP):** Use interfaces to define contracts between services, allowing them to be extended without modification.
3. **Liskov Substitution Principle (LSP):** Ensure that services can be swapped in and out without affecting the correctness of the system.
4. **Interface Segregation Principle (ISP):** Define minimal interfaces for services to reduce dependencies and facilitate testing.
5. **Dependency Inversion Principle (DIP):** Use dependency injection to manage dependencies and promote modularization.

For example, consider a distributed system that includes a user service, an inventory service, and an order service. Each service should have a single responsibility, such as managing users, inventory, and orders respectively. The services should communicate with each other through interfaces, allowing them to be extended without modification. The interfaces should define minimal contracts, reducing dependencies and making testing easier. Finally, we can use dependency injection to manage dependencies between services, promoting modularity and scalability.

30. How would you design a system that adheres to SOLID principles and allows for easy experimentation and prototyping?

To design a system that adheres to SOLID principles and allows for easy experimentation and prototyping, one approach is to use a modular and layered architecture with well-defined interfaces. Each module should have a clear responsibility and be decoupled from other modules, allowing for easy substitution and testing. Dependency injection can be used to facilitate the substitution of

implementations. In addition, feature flags and toggles can be used to enable or disable experimental features without affecting the stability of the system.

31. How would you identify if a codebase violates SOLID principles due to its design patterns?

If a codebase is violating SOLID principles due to its design patterns, it can be identified by analyzing the code for code smells or anti-patterns. For example, if there is a high degree of coupling between classes, or if a class has multiple responsibilities, it could be violating the SRP. Similarly, if a class is hard to extend or modify without affecting other parts of the code, it could be violating the OCP. By analyzing the code in this way, it is possible to identify areas that need improvement to bring the codebase back in line with SOLID principles.

32. How do you balance SOLID principles with the need for performance optimization in a high-performance application?

When optimizing for performance in a high-performance application, it is important to balance SOLID principles with performance considerations. One approach is to use SOLID principles as a guideline and make performance optimizations only when necessary, while measuring the impact of these optimizations on maintainability and scalability. For example, using the strategy pattern can add an extra layer of abstraction, which might have an impact on performance. However, if performance is a critical concern, the decorator pattern can be used instead, which achieves the same goal while minimizing the performance impact.

33. How do you ensure adherence to SOLID principles when integrating third-party libraries into your codebase?

When integrating third-party libraries into your codebase, it's important to ensure that they adhere to SOLID principles. One way to do this is to use dependency injection to decouple your code from the library implementation. This allows you to swap out the library with minimal impact on your code. Additionally, you can wrap the third-party library in an adapter that conforms to the SOLID principles. This way, you can maintain the separation of concerns and avoid coupling your code to the specific implementation of the library.

Source of questions:

1. https://www.adaface.com/blog/solid-principles-interview-questions/#q13
2. https://www.qfles.com/interview-question/solid-principles-interview-questions
3. https://climbtheladder.com/solid-design-principles-interview-questions/