Solid Principles PPTX NOTES

19 September 2024 23:33

Introduction to SOLID Principles

- **What is SOLID?**
  - A set of five design principles for object-oriented programming.
  - Helps in creating maintainable and scalable software.

1. Single Responsibility Principle (SRP)

- **Definition**: A class should have only one reason to change.
- **Example**:
  - Class handles user registration and email notifications.
  - Solution: Separate into UserRegistration and EmailService.
- **Key Point**: Each class should focus on one task.

In other words, each class should focus on a single task. For example, if we have a class that handles both user registration and sending emails, we might run into problems when we need to change one of those functionalities. Instead, we can separate this into two classes: UserRegistration for registration and EmailService for sending emails. This makes our code cleaner and easier to manage.

Example: Restaurant Management

- **Explanation:** Consider a restaurant management system. If you have a Menu class that handles both the menu items and the billing process, it violates SRP. Instead, create two separate classes: Menu for managing food items and Bill for handling billing. This makes it easier to update menu items without affecting the billing logic.

2. Open/Closed Principle (OCP)

- **Definition**: Software entities should be open for extension, but closed for modification.
- **Example**:
  - Bonus calculation for employees.
  - Create new classes for new employee types instead of modifying the existing class.
- **Key Point**: Add new functionality with new classes.

This means we should be able to add new features by creating new classes instead of altering existing ones. For instance, if we have a system calculating bonuses for employees, rather than changing the existing class for different employee types, we can create new subclasses for each type, like PermanentEmployee or TemporaryEmployee. This keeps our original code intact and minimizes the risk of introducing bugs.

Example: Online Shopping Cart

- **Explanation:** Imagine an online shopping cart system. If you want to add new payment methods, like PayPal or cryptocurrency, instead of modifying the existing payment processing code, you can create new classes, such as PayPalPayment and CryptoPayment, that extend the existing Payment class. This allows the system to grow without altering the core functionality.

3. Liskov Substitution Principle (LSP)

- **Definition**: Subtypes must be substitutable for their base types without altering correctness.
- **Example**:
  - Bird class with a Penguin subclass (cannot fly).
  - Ensure subclasses can be used without issues.
- **Key Point**: Derived classes should behave like their base class.

This principle states that subtypes must be substitutable for their base types without altering the correctness of the program. For example, if we have a class Bird, and we create a subclass Penguin, we must ensure that using Penguin in place of Bird doesn't break our code. This means that subclasses should fully adhere to the expectations set by their parent class.

Example: Animal Class Hierarchy

- **Explanation:** Suppose you have a base class called Animal with a method makeSound(). If you create a subclass Dog that implements makeSound() as barking, it works fine. However, if you have another subclass Fish that doesn't make a sound, substituting Fish for Animal would break the expected behavior. To adhere to LSP, either ensure all subclasses can fulfill the contract or redesign the class hierarchy so that the behavior aligns.

4. Interface Segregation Principle (ISP)

- **Definition**: Clients should not be forced to depend on interfaces they do not use.
- **Example**:
  - IMachine with multiple methods (Print, Scan, Fax).
  - Break into smaller interfaces: IPrinter, IScanner.
- **Key Point**: Prefer smaller, specific interfaces.

This means we should design smaller, more specific interfaces rather than one large, all-encompassing one. For example, instead of having an interface called IMachine that has methods for printing, scanning, and faxing, we can split it into IPrinter and IScanner. This way, clients only implement what they need, making the code cleaner and more manageable.

Example: Home Appliances

- **Explanation:** Consider a smart home system with various devices like Light, Fan, and Thermostat. Instead of having a single interface, ISmartDevice, with all functionalities (like turnOn(), turnOff(), setTemperature()), create smaller interfaces: ILight, IFan, and IThermostat. This way, each device only implements the methods relevant to its function, making the code cleaner and easier to manage.

5. Dependency Inversion Principle (DIP)

- **Definition**: High-level modules should not depend on low-level modules. Both should depend on abstractions.
- **Example**:
  - Business logic directly using data access class.
  - Use an interface for data access.
- **Key Point**: Depend on abstractions, not on concrete implementations.

Let's say our business logic directly calls a data access class. This creates a tight coupling between them, making it hard to test and extend. Instead, we can define an interface, say IRepositoryLayer, that our data access class implements. Now our business logic can depend on the interface, allowing us to easily switch implementations without altering the high-level code.

- **Example: Notification System**
  - **Explanation:** Think of a notification system in an application. If the main application logic directly sends email notifications using a specific email service, it creates a tight coupling. Instead, define an interface INotificationService that has a method sendNotification(). Implement this interface in different classes like EmailService and SMSService. Now, the application can use any notification method without being tightly coupled to one specific implementation.

- **Recap of SOLID Principles**:
  - SRP: One responsibility per class.
  - OCP: Open for extension, closed for modification.
  - LSP: Subtypes should replace base types without issues.
  - ISP: Use smaller interfaces.
  - DIP: Depend on abstractions.
- **Importance**: These principles help in building flexible, maintainable, and testable software.

o wrap up, we've covered the five SOLID principles: SRP, OCP, LSP, ISP, and DIP. These principles guide us in writing clean, flexible, and maintainable code. By following them, we can enhance our software design and reduce potential issues down the line. Remember, applying these principles will not only help us as developers but also improve the overall quality of our projects.