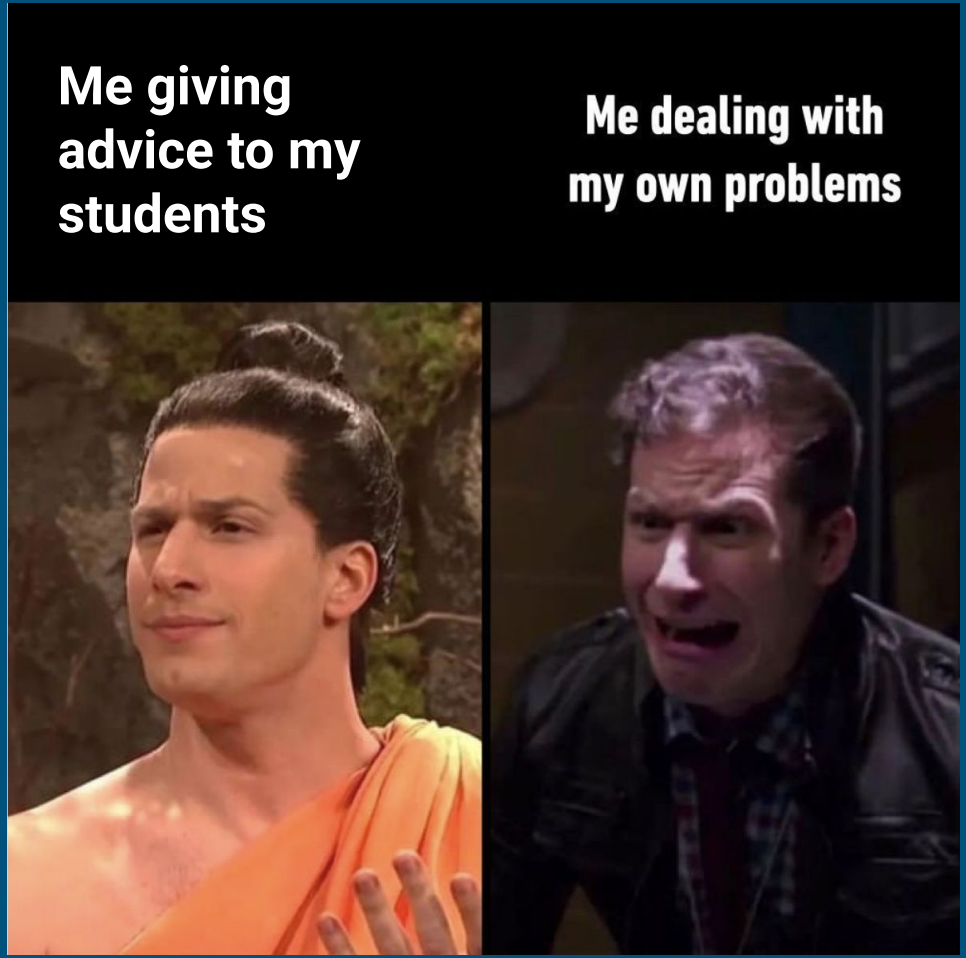
Software Design- SOLID Advice

Manas Jyoti Das, PhD
Computer Science


Me giving advice to my students
Me dealing with my own problems

SOLID

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

SOLID Advice

- SOLID principles, very important design principles, particularly for object oriented programming.
- Each principle has nuances and deeper considerations depending on your specific project and context.

SOLID Principle

- Single Responsibility Principle (SRP)
- A class, module, or function should have only one reason to change. In other words, it should have a specific, focused responsibility
- Original: A Product class handles storing product information (name, price, description), adding to cart, checking stock, and generating product listings.
- Improved: Separate classes: ProductInformation, ShoppingCart, InventoryManager, and ProductListingGenerator. Each focuses on a specific responsibility, making the code more maintainable.

CODE...

SOLID Principle

- Open-Closed Principle (OCP)
- Software entities (classes, modules, functions) should be open for extension but closed for modification. This means you can add new functionality without changing existing code
- You can make changes to a class or function to make it efficient but not to add more functionality. If it is already doing its intended functionality

SOLID Principle

Original: The PaymentProcessor class directly handles credit card and PayPal payments. Adding new payment methods would require modifying this class.
Improved: Define a PaymentInterface with methods like processPayment. Create concrete implementations for CreditCardProcessor, PayPalProcessor, etc. Adding new payment methods involves creating another implementation without modifying existing code.

CODE

SOLID Principle

- Liskov Substitution Principle (LSP)
- Subtypes (derived classes) should be substitutable for their base types without altering program behavior. In simpler terms, if you have a function expecting a base class object, any subclass object should work seamlessly when passed in.

SOLID Principle

- Original: A DiscountableProduct subclass inherits from Product and implements a getDiscount method. However, some products like subscriptions shouldn't offer discounts.
- Improved: Introduce a DiscountedProduct interface inherited by eligible products. The DiscountableProduct subclass now implements both Product and DiscountedProduct, ensuring correct behavior even with new product types

CODE

SOLID Principle

- Interface Segregation Principle (ISP)
- Clients (components of your program) should not be forced to depend on interfaces they don't use. In other words, avoid large, general-purpose interfaces that bundle unrelated functionalities
- Consider an interface AnimalActions with methods like makeSound, eat, and fly. If a class like Snake doesn't fly, it shouldn't be forced to implement the unnecessary fly method. Separate interfaces like SoundMaker, Eater, and Flyer cater to specific functionalities, improving code flexibility

CODE

SOLID Principle

- Dependency Inversion Principle (DIP)
- High-level modules should not depend on low-level modules. Both should depend on abstractions (interfaces). Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions. This promotes loose coupling and testability.
- Example: Suppose your program interacts with different databases (low-level modules). Instead of hardcoding specific database calls in high-level code, define an abstraction like DatabaseInterface. Different concrete database implementations can then inherit from this interface, allowing you to easily switch between databases without impacting high-level logic.

Code