Tired of SOLID examples featuring Animal, Dog, and Cat?

Same here.

Here are SOLID examples from real .NET apps:

Kristijan Kralj

SINGLE RESPOSIBILITY PRINCIPLE

Each class should have a single, well-defined responsibility.

Example where this principle is broken:

Service God class. A service class that takes care of all responsibilities for a single entity...

SINGLE RESPOSIBILITY PRINCIPLE

```
public class UserService
{
    // 1. Business logic: Creating and managing users
    public void CreateUser(string username, string password)
    {
    }

    public void DeleteUser(int userId)
    {
        // Logic to delete user
    }

    // 2. Validation logic (should be delegated to a separate class)
    public bool ValidatePassword(string password)
    {
    }

    // 3. Data access (should be handled by a repository)
    public User GetUserById(int userId)
    {
    }

    // 4. Email/Notification logic (should be handled by a separate service)
    public void SendWelcomeEmail(string emailAddress)
    {
    }

    // 5. Logging (should be handled by a logging framework/service)
    public void LogUserAction(int userId, string action)
    {
    }
}
```

SOLUTION: SPLIT INTO SEVERAL SMALLER CLASSES

```
public class UserRepository
{
    public void Add(User user)
    {
    }

    public void Delete(int userId)
    {
    }

    public User GetById(int userId)
    {
    }
}

public class PasswordValidator
{
    public bool IsValid(string password)
    {
    }
}

public class EmailSender
{
    public void SendWelcomeEmail(string emailAddress)
    {
    }
}

public class UserLogger
{
    public void LogAction(int userId, string action)
    {
    }
}
```

OPEN/CLOSED PRINCIPLE

You should be able to add new functionality without changing existing code. You can achieve it through composition.

Example where this principle is broken?

OPEN/CLOSED PRINCIPLE

A way to configure EF Core classes is to use ModelBuilder.

You can do it in OnModelCreating method of your DbContext.

But this violates the open-closed principle because every entity change also changes the DbContext class:

OPEN/CLOSED PRINCIPLE

```
public class AppDbContext : DbContext
{
    public DbSet<SocialMediaAccount> SocialMediaAccounts { get; set; }
    public DbSet<User> Users { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Configuration for SocialMediaAccount is mixed directly here
        modelBuilder.Entity<SocialMediaAccount>(builder =>
        {
            builder.HasKey(s => s.Id);

            builder.Property(s => s.Platform)
                .IsRequired()
                .HasMaxLength(50);

            builder.Property(s => s.Username)
                .IsRequired()
                .HasMaxLength(100);

            builder.HasOne(s => s.User)
                .WithMany(u => u.SocialMediaAccounts)
                .HasForeignKey(s => s.UserId);
        });

        // Imagine 100 more entities configured the same way...
    }
}
```

SOLUTION: USE IENTITYTYPECONFIGURATION

```
public class SocialMediaAccountConfiguration : IEntityTypeConfiguration<SocialMediaAcco
{
    public void Configure(EntityTypeBuilder<SocialMediaAccount> builder)
    {
        builder.ToTable("SocialMediaAccounts");

        builder.HasKey(s ⇒ s.Id);

        builder.Property(s ⇒ s.Platform)
            .IsRequired()
            .HasMaxLength(50);

        builder.Property(s ⇒ s.Username)
            .IsRequired()
            .HasMaxLength(100);

        builder.HasOne(s ⇒ s.User)
            .WithMany(u ⇒ u.SocialMediaAccounts)
            .HasForeignKey(s ⇒ s.UserId);
    }
}
```

NOW, DBCONTEXT TURNS INTO THIS:

```
public class AppDbContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Automatically apply all IEntityTypeConfiguration<T> in this assembly
        modelBuilder.ApplyConfigurationsFromAssembly(typeof(AppDbContext).Assembly);
    }
}
```

And now you can add as many entities as you want without changing the DbContext every time a new entity is added or changed.

LISKOV SUBSTITUTION PRINCIPLE

You should be able to swap base and child classes without affecting how the application works.

Explanation:

- This principle is about inheritance.
- Avoid inheritance in C# code.

Let’s move on.

INTERFACE SEGREGATION PRINCIPLE

Large, bloated interfaces should be split into smaller, specific ones.

How I like to explain this principle: Interface Segregation Principle is the Single Responsibility principle, but for interfaces.

Example where this principle is broken?

INTERFACE SEGREGATION PRINCIPLE

In the same way you have UserService, the IUserService interface carries the method signatures:

```
public interface IUserService
{
    void CreateUser(string username, string password);
    void DeleteUser(int userId);
    bool ValidatePassword(string password);
    User GetUserById(int userId);
    void SendWelcomeEmail(string emailAddress);
    void LogUserAction(int userId, string action);
}
```

SOLUTION:

Split the interface into several smaller ones:

- IUserRepository
- IPasswordValidator
- IEmailSender
- ILogger

Or you can entirely skip some of them.

DEPENDENCY INVERSION PRINCIPLE

Business logic layers should not depend on infrastructure layers.

Code should depend on interfaces or abstract classes, not concrete implementations.

Ok, this principle can be observed in 2 ways.

1. BUSINESS LOGIC LAYERS SHOULD NOT DEPEND ON INFRASTRUCTURE LAYERS.

If you use Clean Architecture, then the Entities and Use Cases, aka business layer (domain logic), don’t depend on the infrastructure layers, aka database, file system, external APIs, etc.

(Combine it with Vertical Slice Architecture for best results.)

2. CODE SHOULD DEPEND ON INTERFACES OR ABSTRACT CLASSES.

This is not only about using IWhatever interface and inject it into your classes.

It’s also about recognizing what non-deterministic .NET built-in classes you have in your code.

And abstracting them so you can unit test your code more easily.

2. CODE SHOULD DEPEND ON INTERFACES OR ABSTRACT CLASSES.

The two most obvious suspects are:

1. DateTime.Now -> I abstract it with IDateTimeProvider.
2. Guid.NewGuid -> I abstract it with IGuidGenerator.

Then, in my tests, I have fake implementations of these interfaces, and all tests are deterministic.

So there you have it.

SOLID principles using a real-world .NET code.

Let me know if this clarifies SOLID for you.

WANT TO PROGRESS YOUR CAREER TO A MORE SENIOR LEVEL?

Every Friday, my newsletter provides actionable advice on architecting and building scalable, maintainable, and testable .NET applications.

LINK IN BIO >>>