JavaScript Design Patterns & SOLID Principles Presentation

Presented by
Mohamed Seleem

What Are Design Patterns ?

- Design Patterns are smart solutions that developers use to fix repeating design problems.
- They give us a standard way to write code that’s easy to change and understand.
- You can think of them like templates or ready-made solutions for common situations.

Why We Use Design Patterns ?

Before using patterns, developers had many problems:

- Code was messy and repeated everywhere.
- Changing one thing broke other parts.
- Projects were hard to scale and maintain.

✅ With Design Patterns, code becomes organized, flexible, and reusable.

Categories of Design Patterns

Design patterns are usually divided into three main categories:

- Creational Patterns – Focus on how we create objects.
- Structural Patterns – Focus on how we connect and organize objects.
- Behavioral Patterns – Focus on how objects communicate and work together.

Creational Patterns

Creational patterns help control object creation so the developer doesn’t repeat the same logic every time.

Some common examples are:

- Constructor Pattern: Helps create and initialize objects easily.
- Singleton Pattern: Ensures only one instance of a class exists in the program.
- Factory Pattern: Creates objects without exposing the creation logic.
- Builder Pattern: Builds complex objects step by step.
- Dependency Injection: Passes needed objects to a class instead of creating them inside it

Structural Patterns

Structural patterns are about how objects are connected or combined to form larger structures. They help reduce tight connections between components and make code easier to expand later.

Behavioral Patterns

Behavioral patterns describe how objects communicate and work together to complete a task.
Two of the most common behavioral patterns are Strategy Pattern and Observer Pattern.

Strategy Pattern

The Strategy Pattern allows us to switch between different algorithms or behaviors easily.
Instead of writing many if/else statements, we separate each behavior into its own class.
This makes the code cleaner and more flexible.
For example, we can create different discount strategies like student discount, member discount, or no discount — and change them anytime without touching the main code.

Observer Pattern

The Observer Pattern is used when one object needs to notify others automatically when something changes.
For example, in social media, when someone posts something new, all followers get a notification instantly.
This pattern reduces the connection between classes and makes the system more dynamic and responsive.

Transition to SOLID Principles

Now, let’s move to SOLID Principles, which are another set of guidelines for writing clean and well-structured code.
While design patterns are about solving specific problems, SOLID principles are about the general design of the system.

What Are SOLID Principles

SOLID is a group of five object-oriented design principles that help developers create software that is easy to maintain, test, and extend.

The five principles are:

1. Single Responsibility Principle (S)
2. Open/Closed Principle (O)
3. Liskov Substitution Principle (L)
4. Interface Segregation Principle (I)
5. Dependency Inversion Principle (D)

Why Do We Need SOLID Principles

When developers ignore structure, the project becomes messy, hard to change, and full of bugs.
SOLID helps prevent these problems by giving us clear rules for designing good classes and relationships between them.

Single Responsibility Principle

This principle means that each class should have only one reason to change.
In other words, a class should do only one main job.
For example, if you have a class that handles both user login and report creation, that's wrong.
You should separate them into two classes — one for authentication and another for generating reports.
This makes your code cleaner and easier to maintain.

Open/Closed Principle

This principle says that a class should be open for extension but closed for modification.
That means we can add new functionality without changing the existing code.
For example, if we have a payment system, instead of editing the same payment class every time we add a new method like PayPal or Stripe, we can simply create new subclasses that extend the main payment behavior.
This keeps the original code safe and stable.

Liskov Substitution Principle

This principle says that a class should be open for extension but closed for modification.
That means we can add new functionality without changing the existing code.
For example, if we have a payment system, instead of editing the same payment class every time we add a new method like PayPal or Stripe, we can simply create new subclasses that extend the main payment behavior.
This keeps the original code safe and stable.

Interface Segregation Principle

This principle means that classes should not be forced to implement methods they don’t use.
Instead of having one large interface, it’s better to have multiple small, specific ones.
This keeps each class focused and simple.

Dependency Inversion Principle

This principle says that high-level modules should not depend on low-level modules, but both should depend on abstractions.

This means we should depend on interfaces or abstract classes instead of concrete implementations.

It makes the code more flexible and easier to test or change in the future.