DESIGN PATTERN

- **A DESIGN PATTERN IS A TERM USED FOR A GENERAL, REUSABLE SOLUTION TO A COMMONLY OCCURRING PROBLEM IN SOFTWARE DESIGN.**

- **DESIGN PATTERNS ARE REUSABLE SOLUTIONS TO COMMONLY OCCURRING PROBLEMS IN SOFTWARE DESIGN.**

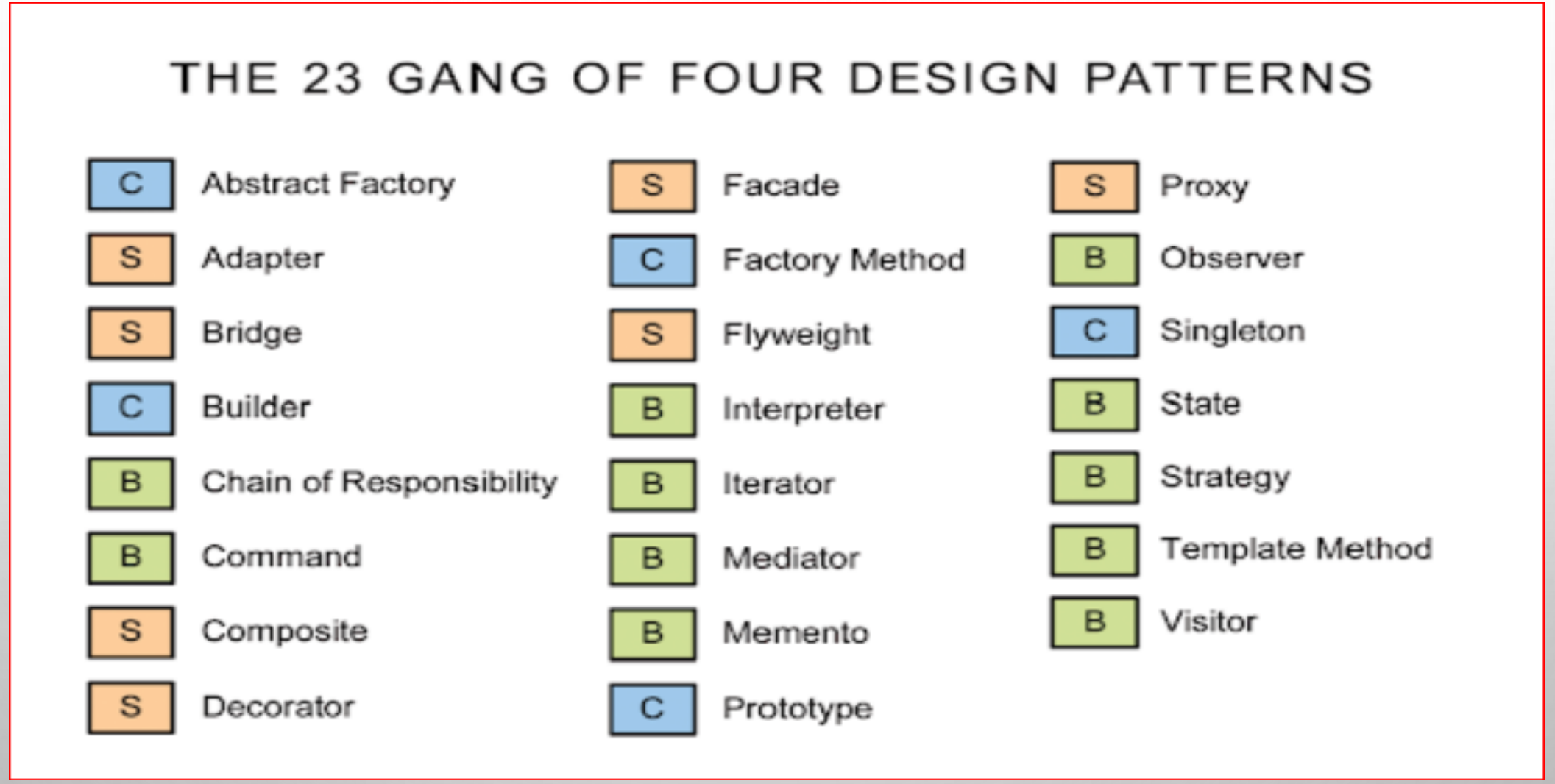
- **IT IS NOT A FINISHED PIECE OF CODE THAT CAN BE DIRECTLY APPLIED TO YOUR PROGRAM. BUT RATHER, IT IS MORE LIKE A TEMPLATE OR DESCRIPTION THAT CAN GIVE YOU AN IDEA OF APPROACHING A PROBLEM AND INSPIRING SOLUTIONS.**

WHY TO USE DESIGN PATTERN ?

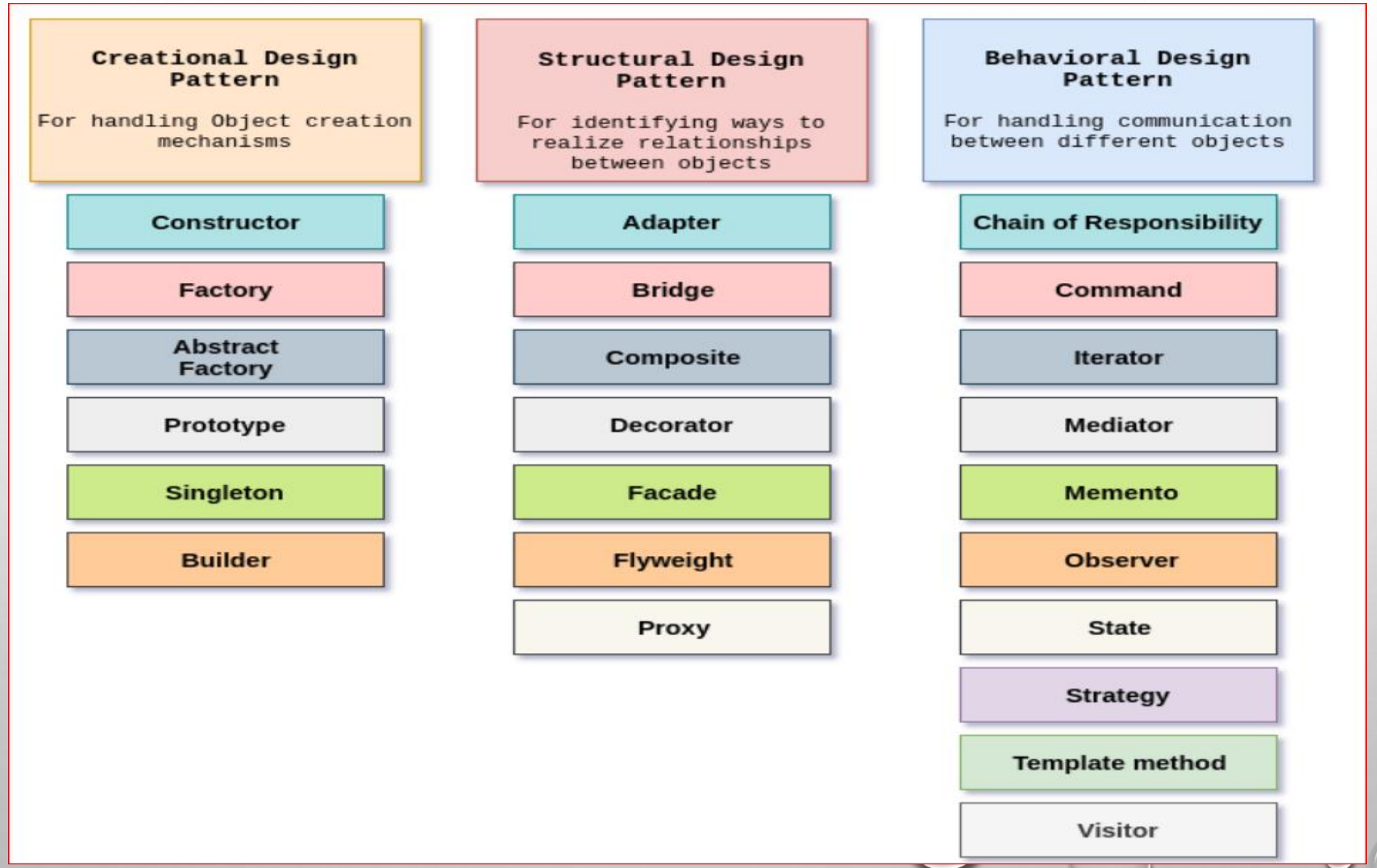
The main benefits we get from design patterns are the following:

- **They are proven solutions:** Because design patterns are often used by many developers, you can be certain that they work. And not only that, you can be certain that they were revised multiple times and optimizations were probably implemented.
- **They are easily reusable:** Design patterns document a reusable solution which can be modified to solve multiple particular problems, as they are not tied to a specific problem.
- **They are expressive:** Design patterns can explain a large solution quite elegantly.
- **They ease communication:** When developers are familiar with design patterns, they can more easily communicate with one another about potential solutions to a given problem.
- **They prevent the need for refactoring code:** If an application is written with design patterns in mind, it is often the case that you won't need to refactor the code later on because applying the correct design pattern to a given problem is already an optimal solution.
- **They lower the size of the codebase:** Because design patterns are usually elegant and optimal solutions, they usually require less code than other solutions.

23 DESIGN PATTERNS IN ALPHABETICAL ORDER



Design Pattern Categorization



CREATIONAL:

Creational	Based on the concept of creating an object.
Class	
Factory Method	This makes an instance of several derived classes based on interfaced data or events.
Object	
Abstract Factory	Creates an instance of several families of classes without detailing concrete classes.
Builder	Separates object construction from its representation, always creates the same type of object.
Prototype	A fully initialized instance used for copying or cloning.
Singleton	A class with only a single instance with global access points.

STRUCTURAL :

Structural	Based on the idea of building blocks of objects.
Class	
Adapter	Match interfaces of different classes therefore classes can work together despite incompatible interfaces.
Object	
Adapter	Match interfaces of different classes therefore classes can work together despite incompatible interfaces.
Bridge	Separates an object's interface from its implementation so the two can vary independently.
Composite	A structure of simple and composite objects which makes the total object more than just the sum of its parts.
Decorator	Dynamically add alternate processing to objects.
Facade	A single class that hides the complexity of an entire subsystem.
Flyweight	A fine-grained instance used for efficient sharing of information that is contained elsewhere.
Proxy	A place holder object representing the true object.

BEHAVIORAL:

Behavioral	Based on the way objects play and work together.
Class	
Interpreter	A way to include language elements in an application to match the grammar of the intended language.
Template Method	Creates the shell of an algorithm in a method, then defer the exact steps to a subclass.
Object	
Chain of Responsibility	A way of passing a request between a chain of objects to find the object that can handle the request.
Command	Encapsulate a command request as an object to enable, logging and/or queuing of requests, and provides error-handling for unhandled requests.
Iterator	Sequentially access the elements of a collection without knowing the inner workings of the collection.
Mediator	Defines simplified communication between classes to prevent a group of classes from referring explicitly to each other.
Memento	Capture an object's internal state to be able to restore it later.
Observer	A way of notifying change to a number of classes to ensure consistency between the classes.
State	Alter an object's behavior when its state changes.
Strategy	Encapsulates an algorithm inside a class separating the selection from the implementation.
Visitor	Adds a new operation to a class without changing the class.

Follow me for such info

https://www.linkedin.com/in/priya-bagde/

GitHub

https://github.com/priya42bagde

YouTube

https://www.youtube.com/channel/UCK1_Op30_pZ1zBs9l3HNyBw/videos (Priya Frontend Vlogs)