SOLID Principles: five essential guidelines that enhance software design, making code more maintainable and scalable.

S- single Responsibility principle:

* A class should have only one reason to change
* Eg. I*magine a baker who is responsible for baking bread. The baker’s role is to focus on the task of baking bread, ensuring that the bread is of high quality, properly baked, and meets the bakery’s standards.*

O- Open/ closed principle

* Open for extension but closed for modification
* you should be able to extend a class behavior, without modifying it.
* Eg. I*magine you have a class called PaymentProcessor that processes payments for an online store. Initially, the PaymentProcessor class only supports processing payments using credit cards. However, you want to extend its functionality to also support processing payments using PayPal.*
* To add new paymaent process create PayPalPaymentProcessor which extends PaymentProcessor class

L- Liskov’s substitution principle: introduced by Barbara liskov in 1987

* Derived or child classes must be substitutable for their base or parent classes
* If class B is subtype of Class A, then we should be able to replace object of A with B without breaking the behavior of the program
* Subclass should extend the capability of parent class not narrow it down.
* Eg. *examples of this principle is a rectangle having four sides. A rectangle’s height can be any value and width can be any value. A square is a rectangle with equal width and height. So we can say that we can extend the properties of the rectangle class into square class.*

I- Interface segmented/ Segregation principle

* Interfaces should be such that client should implement unnecessary functions they do not need
* *Eg.Suppose if you enter a restaurant and you are pure vegetarian. The waiter in that restaurant gave you the menu card which includes vegetarian items, non-vegetarian items, drinks, and sweets.*

D- Dependency inversion principle:

* Class should depend on interfaces rather than concrete classes
* High-level modules should not depend on low-level modules. Both should depend on abstractions
* Eg. I*n a software development team, developers depend on an abstract version control system (e.g., Git) to manage and track changes to the codebase. They don’t depend on specific details of how Git works internally.*

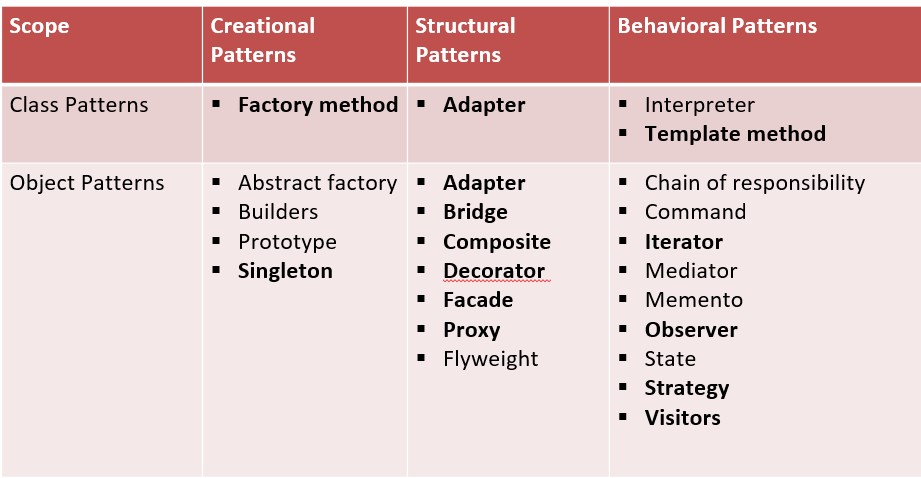
Advantages:

* Avoid duplicate code
* Easy to maintain
* Easy to understand
* Flexible software
* Reduce Complexity
* Helps to reduce tight coupling

**Design pattern in java**

* Design pattern: A reusable and named solution to recurring problem in a context.
* Pattern Classification:

1. Purpose:
2. Creational: Used for creating or instantiating objects and classes.
3. Abstract Factory
4. Builder
5. factory method
6. Singleton
7. Prototype
8. Structural: Used for structuring more than one classes or objects together. Structuring around object and classes comes in this category like inheritance, segregation.
9. Adapter(class)
10. Adapter(object)
11. Bridge
12. Composite
13. Decorator
14. Façade
15. Flyweight
16. Proxy
17. Behavioral: Used for identifying and setting up common communication patterns among objects.
18. Interpreter
19. Template Method
20. Chain of responsibility
21. Command
22. Iterator
23. Mediator
24. Memento
25. Observer
26. State
27. Strategy
28. scope: 1. Class: Factory method, Adapter(Class), Interpreter, Template method , 2. Object



Explanation

1. Factory Method Design: Useful for creation of objects that fall under same categorization but still have different properties

* Creational: used to create object of similar type
* Goal: to hide complexity of object creation
* Real-world example and application: Architects create design and interfaces for team
* Eg. Calender and NumberFormat class in java example of factory method pattern
* Eg. Frameworks and library
* Pros: Guarantees abstraction, Code is flexible and adaptable, Very useful for frameworks and libraries
* Cons: Complex code, takes time to set the base, Not a pattern that can be refactored into.

1. Builder Design pattern

* Creational: creation of complex objects
* Helps with Immutable classes where less need for exposing setters of class
* Use telescope constructor but does not solve problems 100% for having multiple combinations so we use builder patterns create static inner class and then return it in constructor
* Pros: handles complexity, can be refactored into
* Cons: class instance return is immutable, uses inner static class

1. Abstract Factory Method Design pattern:

* Factory of factory pattern
* Means pattern inside a pattern
* Creational: Creating objects which belongs to a family of similar objects
* Implemented using a common interface ( implementation is deferred to sub-classes/ concrete classes)
* Example: Java -DocumentBuilder method
* Pros: Good for abstraction and family of similar objects, Loose coupling between client and actual/concrete code, all classes follow single responsibility principle, supports open closed principle
* Cons: code becomes complicated/ complex, pattern inside a pattern

1. Singleton Design pattern:

* It solves problem like when only one instance of a class is needed(shared resource) across the application.
* Make the variable initialized by a private constructor and access to be given only by a getter() method
* Note: Singleton class never accepts parameter. If it accepts parameter, then it becomes a factory. So avoid it.
* Provides thread safety.
* Eager loading: the instance is already initialized as soon as the application is up. Utilizes more memory, works well with one singleton class
* Lazy loading: The instance is initialized only when any app module calls for it. Less memory utilization, Suitable for multiple singleton classes
* Pros: Neat way to handle access to shared global resource, only one instance,
* Cons; avoid Used with parameters and confused with factory, Thread safety has to be insured else can be dangerous., hard to write unit tests.

1. Adapter Design pattern

* It is structural design pattern
* Example: Java Stream Classes
* We can use when interfaces are not compatible with other

<https://dev.to/somadevtoo/top-10-design-patterns-for-programming-interviews-8j4>