Catalog of Design Patterns

Designed by Rita Ganatra

Catalog of Design Patterns

The Catalog contains 23 design Patterns:

- **Abstract Factory:** It Indicates what factory is to be instantiated, **provides an interface to create families of objects(related / dependent)** without any specification of their concrete classes.
- **Adaptor:** Id Adapt or **converts an interface of a class into another one** according to the client expectation and hence, overcomes the problem of incompatible interfaces thereby **enabling the classes to work together.**
- **Bridge:** It Separates **abstraction from its implementation** to make them independent.

- **Builder:** It Separates the **complex objects constructions** from their representation in order to create different representations with the same construction process.
- **Chain Of Responsibility:** It Enables the handling of command objects by passing them to other objects by using the logic present in the processing of objects. In other words, Its **decouples sender and receiver** by formatting a chain of receiving objects to pass the request until the request is handled by an object.
- **Command:** It encapsulates the action and its parameters and hence, **enables to parameterize** the different requests of the clients such as **long or queue requests**. It also assists undoable operations.

- **Composite:** It represents the objects in a **tree structure** where each **object represents the same interface.** This enables clients to treat individual objects and their compositions uniformly.
- **Decorator: It Adds additional functionality to a class at runtime**. This Enables flexibility to subclass for adding functionality.
- **Facade:** It **creates a simplified/unified interface** of existing interfaces in the subsystems so as to handle common tasks easily.

- **Factory Method:** It Focuses on **objects creation** of specific implementation. lets the subclass decide as to which class to be instantiated.
- **Flyweight:** It Performs sharing of **common objects properties** by a large number of objects to save space.
- **Interpreter:** It Deals with the **implementation** of a specified computer language that solves specific problems. It interprets sentences in language by representing the grammar of language along with an interpreter.

- **Iterator:** It Enables sequential aggregate objects elements by **hiding their underlying representations.**
- **Mediator:** It provides a **unified interface** to the set of interfaces in a subsystem. It provides loose coupling
- **Momento:** It supports the **rollback mechanism** by enabling the objects, to restore to their previous state without violation of encapsulation.

- Observer: Whenever a**n object changes its state, it raises an event that notifies other objects and updates them automatically.** This defines a one-to-many dependency between the objects.
- Prototype: Here Prototypical instance determines the **type of objects** to be created. Further new objects are created by cloning(duplicating) this prototype.
- proxy: It provides an illusion by **applying placeholder to other objects** in order to have control over it.

- **Singleton:** It Provides **restrictions on instantiating a class to a single object** and also makes it globally accessible.
- **State:** It **Permits an alteration in the object’s behavior** with alteration in its state .i.e allows objects type to change at runtime.
- **Visitors:** It Describes the skeleton(basic structure) of a program, enables subclasses to define some steps of the algorithm, and also to redefine certain steps without affecting the structure of the algorithm.
- **Strategy:** It defines the algorithms and their selection based upon the clients.

- **Template Method** : Define the **skeleton of an algorithm** in an operation, deferring some steps to subclasses. Template Method lets **subclasses redefine certain steps of an algorithm** without changing the algorithm’s structure.
- **Visitor**: **Represent an operation to be performed** on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Uses of Design Patterns in Software Engineering:

A design pattern in the context of software engineering is a template or **reusable solution to common problems occurring in software design**. This is usually represented with the classes and objects, interfaces, etc. The one application of a design pattern is “Reusability of Solutions”, By using the proven solution, users can solve software development issues that enable the development of highly cohesive modules with coupling.

Need of Design pattern:

Here are some key reasons why design patterns are important:

- ➢ **Reusability**: Design patterns provide **reusable solutions** to common problems, reducing the need to reinvent the wheel and enabling code reuse across different projects.
- ➢ **Efficiency**: By using well-established patterns, developers can **save time** and effort, **speeding up the development process and increasing productivity.**
- ➢ **Scalability**: Design patterns help create **scalable and maintainable code** that can grow with the project, making it easier to add new features and handle increased complexity.

- **Readability**: Patterns promote a shared vocabulary among developers, **improving communication** and understanding of the codebase.
- **Best Practices**: Design patterns encapsulate best practices and proven techniques, helping developers avoid common pitfalls(mistakes) and build robust software.
- **Flexibility**: Patterns provide flexible solutions that can be adapted to different situations, making the code more adaptable to change.
- **Maintainability**: Well-structured code using design patterns is easier to maintain, debug, and refactor, leading to a more sustainable codebase.
- **Collaboration**: Design patterns facilitate collaboration among developers by providing clear and consistent approaches to solving common problems

Thank You