Introduction to Design Patterns

ddas.tech/introduction-to-design-patterns/

October 27, 2022

Created By: Debasis Das (Oct 2022)

What is a Design Pattern?

- A design pattern represents a reusable solution to a commonly occurring problem in software design.
- A design pattern is a template for a design that solves a general, recurring problem in a particular context
- Design patterns are proven solutions under certain software requirements. Such design patterns improve the body of knowledge of software design, so that proven solutions can be adopted with required modifications (if needed) in solving a new problem.

Benefits of using design patterns include:

- Improved modularity
- Better testability using known techniques
- Assuring that software quality attributes that can be derived from known patterns
- Maintainability

Types of Design Pattern

(GoF) design patterns are considered as widely used and known design patterns in many object- oriented systems

Design Patterns are classified into 3 categories namely

- **Creational Design Patterns**
- **Structural Design Patterns**
- **Behavioral Design Patterns**

Creational Design Pattern

- **Abstract Factory**
  - Abstract Factory provides an interface for creating families of related or dependent objects without specifying their concrete classes.
  - The family of objects created by the factory are defined at run-time

- **Prototype**
    - Prototype design pattern is used to instantiate a new object by copying all the properties of an existing object, creating an independent clone.
    - This practice is useful when the construction of a new object is inefficient
- **Builder**
    - Builder Design Pattern is used for creating complex objects with constituent parts that must be created in the same order or using a specific algorithm
    - An external class controls the construction algorithm
- **Factory Method**

    Rather than the client creating the object instance of different types, it uses the service of an object factory to request the type of object instance needed.
    - Factory Design pattern is used to replace class constructors while abstracting the process of object generation so that the type of object instantiated can be defined at run-time
    - *Example: Java NumberFormat factory class requesting different types of formatting instances NumberFormat Factory*
- **Singleton**
    - Singleton design pattern ensures that only one object of a particular class is ever created.
    - All further references to objects of the singleton class refer to the same underlying instance

Structural Design Pattern

- **Adapter**
    - Adapter design pattern is used to provide a link between two otherwise incompatible types.
    - It converts the interface of a class into another interface that the client expects.
    - It decouples the client from the class of the targeted object
- **Bridge**

    Bridge design pattern is used to separate the abstract elements of a class from the implementation details, providing the means to replace the implementation details without modifying the abstraction
- **Composite**
    - Composite design pattern is used to create hierarchical recursive tree structures of related objects where any element of the structure may be accessed and utilized in a standard manner.
    - The related objects can be used to represent part or whole hierarchies.
    - This pattern allows clients to treat individual objects and composition of objects uniformly

- **Decorator**
    - Decorator design pattern attaches additional responsibilities to an object dynamically.
    - Provides a flexible alternative to subclassing for extending functionalities
    - Similar to subclassing a decorator pattern allows us to incorporate new behavior without modifying existing code.
    - Decorators wrap an object of the class whose behavior they extend.
    - They implement the same interface as the object they wrap and add their own behavior either before or after delegating a task to the wrapped object
- **Facade**
    - Facade pattern is used to define a simplified interface to a more complex subsystem
    - The pattern defines a higher level interface that makes the subsystem easier to use by reducing complexity and hiding communication and dependencies between subsystems
- **Flyweight**

    Flyweight pattern is used to reduce the memory and resource usage for complex models containing many hundred thousands of similar objects
- **Proxy**
    - Proxy pattern is used to provide a surrogate or placeholder object, which references an underlying object.
    - The proxy provides the same public interface as the underlying subject class.
    - We use this pattern to create a representative, or proxy, object that controls access to another object, which may be remote, expensive to create, or in need of securing.
    - This pattern is structurally similar to the Decorator pattern but it serves a different purpose; Decorator adds behavior to an object whereas Proxy controls access to an object.

Behavioral Design Pattern

- **Chain of Responsibility**
    - Chain of responsibility pattern is used to process varied requests, each of which might be dealt by a different handler
    - Decouples the sender of a request from its receiver by giving more than one object a chance to handle the request.
    - The pattern chains the receiving objects together and passes the request along the chain until an object handles it.
    - Each object in the chain either handles the request or passes it to the next object in the chain.

- **Delegate Design Pattern**
  - The delegate design pattern is a common way to establish communication between objects in an application. It is a behavioral pattern, which means it focuses on how objects collaborate with each other to perform tasks.
  - In the delegate pattern, one object (the delegating object) delegates a task or responsibility to another object (the delegate) to handle it on its behalf. The delegating object retains a reference to its delegate and calls its methods when it needs to communicate information or ask for a response.
  - Details about the Delegate Design Pattern and Sample Implementation in Swift can be found here https://ddas.tech/delegate-design-pattern-in-swift/
- **Memento**

  The memento pattern is used to capture the current state of an object and store it in such a manner that it can be restored at a later time without breaking the rules of encapsulation
- **Command**
  - Command pattern is used to express a request, including the call to be made and all of its required parameters in a command object. The command may then be executed immediately or held for later use.
  - The request object binds together one or more actions on a specific receiver.
  - The Command pattern separates an object making a request from the objects that receive and execute that request.
- **Observer**
  - Observer Pattern is used to allow an object to publish changes to its state. Other objects subscribe to be immediately notified of any changes
  - The Observer design pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  - The Observer pattern is essentially a publish-and-subscribe model in which the subject and its observers are loosely coupled.
  - Communication can take place between the observing and observed objects without either needing to know much about the other.
  - Here is a post on KVC KVO Design Pattern in Swift
- **Interpreter**

  The interpreter pattern is used to define the grammar for instructions that form part of a language or notation, whilst allowing the grammar to be easily extended.
- **State**
  - The state pattern is used to alter the behaviour of an object as its internal state changes.
  - The pattern allows the class for an object to apparently change at run-time.

- **Iterator**
  - The iterator pattern is used to provide a standard interface for traversing a collection of items in an aggregate object without the need to understand its underlying structure.
  - The Iterator design pattern provides a way to access the elements of an aggregate object (that is, a collection) sequentially without exposing its underlying representation.
  - The Iterator pattern transfers the responsibility for accessing and traversing the elements of a collection from the collection itself to an iterator object.
  - The Iterator defines an interface for accessing collection elements and keeps track of the current element.
- **Strategy**

  Strategy pattern is used to create an interchangeable family of algorithms from which the required process is chosen at run-time.
- **Template Method**

  The Template Method design pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. The Template Method pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- **Mediator**
  - Mediator pattern is used to reduce coupling between classes that communicate with each other. Instead of classes communicating directly, and thus requiring knowledge of their implementation, the classes send messages via a mediator object.
  - The Mediator design pattern defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
  - A "mediator object" in this pattern centralizes complex communication and control logic between objects in a system. These objects tell the mediator object when their state changes and, in turn, respond to requests from the mediator object.
- **Visitor**

  Visitor pattern is used to separate a relatively complex set of structured data classes from the functionality that may be performed upon the data that they hold.

Reference

Gang of Four : Design Patterns: Elements of Reusable Object-Oriented Software