Design Patterns In PHP Course

PRESENTED BY RAMY HAKAM

1.1 Abstract Factory

PURPOSE

To create series of related or dependent objects without specifying their blueprint classes. Usually the created classes all implement the same interface.

1.2 Builder

PURPOSE

The intent of the Builder design pattern is to separate the construction of a complex object from its representation. By doing so the same construction process can create different representations

1.3 Factory Method

PURPOSE

Factory pattern is one of the most used design patterns
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

1.4 Pool

PURPOSE

Uses a set of initialized objects kept ready to use A client of the pool will request an object from the pool and perform operations on the returned object. When the client has finished, it returns the object, which is a specific type of factory object, to the pool rather than destroying it.

1.5 ProtoType

PURPOSE

In the Prototype Pattern we create one standard object for each class, and clone that object to create new instances even complex ones, without coupling to their specific classes
To avoid the cost of creating objects with the standard way

1.6 Simple Factory

PURPOSE

In this pattern, a class simply creates the object you want to use, This allows interfaces for creating objects without exposing the object creation logic to the client
You can consider this is simple other way from using "new"

1.7 Static Factory

PURPOSE

Similar to the AbstractFactory, This pattern is used to create series of related or dependent objects. The difference between this and the abstract factory pattern is that the static factory pattern uses just one static method to create all types of objects it can create. It is usually named factory or build.

Factory Patterns Variation

ABSTRACT FACTORY:

Create a list of related Objects with one function for each class ,
Can have subclasses, Can be mocked

STATIC FACTORY:

Create a list of related Objects with OLNY ONE Static function for all classes
Only one instance, can not be extended and can not be mocked

Factory Patterns Variation

FACTORY METHOD:

Define an interface for creating an object, but let subclasses decide which class to instantiate.

SIMPLE FACTORY:

A class simply creates the object you want to use, This allows interfaces for creating objects without exposing the object creation logic to the client

Where Is Singleton!

Singleton Pattern is Deprecated!!

Structural Patterns

How Objects Related To Each other!

2.1 Adapter / Wrapper

PURPOSE

The Adapter acts as a wrapper between two objects. It catches calls for one object and transforms them to format and interface recognizable by the second object.
To allows classes to work together that normally could not because of incompatible interfaces by providing its interface to clients while using the original interface

2.2 Bridge

PURPOSE

Decouple an abstraction from its implementation so that the two can vary independently. The bridge pattern allows the Abstraction and the Implementation to be developed independently and the client code can access only the Abstraction part without being concerned about the Implementation part.

Publish abstraction interface in a separate Inheritance hierarchy, and put the implementation In its own inheritance hierarchy.

2.3 Composite

PURPOSE

composite design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

The greatest benefit of this approach is that you don’t need To care about the concrete classes of objects that compose The tree.You can treat them all the same via the common Interface. When you call a method, the objects Themselves pass the request down the tree.

2.4 Data Mapper

PURPOSE

A Data Mapper, is a Data Access Layer that performs bidirectional transfer of data between a persistent data store (often a relational database) and an in memory data representation (the domain layer). Example "Doctrine ORM"

2.5 Decorator

PURPOSE

Decorator is a structural design pattern that lets you attach new behaviours to objects by placing these objects inside special wrapper objects that contain the behaviours.
To dynamically add new functionality to class instances.
in the Runtime not in the compile time.

2.6 Dependency Injection

PURPOSE

Is a software design pattern that allows avoiding hard-coding dependencies and makes possible to change the dependencies both at runtime and compile time

Dependency injection is a technique whereby one object supplies the dependencies of another object.

Using Dependency injection You can write more maintainable, testable code.

2.7 Facade

PURPOSE

Facade provides a simplified interface to a library, a framework, or any other complex set of classes.
Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.
Use the Facade pattern when you need to have a limited but straightforward interface to a complex Subsystem.

2.8 Fluent Interface

PURPOSE

A Fluent Interface or Fluent Builder is an object oriented API that provides "more readable" code.
A fluent interface allows you to chain method calls, which results in less typed characters when applying multiple operations on the same object.
It is a very useful technique to make an interface of a Class more expressive and easier to use for clients

2.8 Proxy Object

PURPOSE

is a structural design pattern that provides an object that acts as a substitute for a real service object used by a client. A proxy receives client requests, does some work (access control, caching, etc.) and then passes the request to a service object.

A proxy controls access to the original object, allowing you perform something either before or after the
 Request gets through to the original object

Behavioural Patterns

How Objects Works together!

3.1 Chain of Responsibility

PURPOSE

Is a behavioural design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.
To build a chain of objects to handle a call in sequential order. If one object cannot handle a call,
It delegates the call to the next in the chain and so forth.

3.2 Command

PURPOSE

is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request.

We have an Invoker and a Receiver. This pattern uses a "Command" to delegate the method call against the Receiver and presents the same method "execute". Therefore, the Invoker just knows to call "execute" to process the Command of the client. The Receiver is decoupled from the Invoker.

3.3 Iterator

PURPOSE

Iterator is a behavioral design pattern that allows sequential traversal through a complex data structure without exposing its internal details.
In other words it’s used to make an object iterable and to make it appear like a collection of objects.
The pattern is very common in PHP code. Many frameworks and libraries use it to provide a standard way for traversing their collections.

3.4 Mediator

PURPOSE

is a behavioral design pattern that reduces coupling between components of a program by making them communicate indirectly, through a special mediator object.
This pattern provides an easy way to decouple many components working together

Mediator is still uses for example like the event dispatchers of many PHP frameworks or some implementations of MVC controllers.

3.5 Memento

PURPOSE

is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

The memento pattern is implemented with three objects

- Memoento: An object that contains a concrete unique snapshot of state of any object or resource
- Originator: – It is an object that contains the actual sta an external object is strictly specified type
- Caretaker: Controls the states history.

3.6 Observer

PURPOSE

is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
To implement a publish/subscribe behaviour to an object, whenever a "Subject" object changes its state, the attached "Observers" will be notified. It is used to shorten the amount coupled objects and uses loose coupling instead.

3.7 Specification

PURPOSE

Builds a clear specification of business rules, where objects can be checked against. The composite specification class has one method called isSatisfiedBy that returns either true or false depending on whether the given object satisfies the specification.

it would be a logic that returns either true or false

3.8 State

PURPOSE

The pattern extracts state-related behaviors into separate state classes and forces the original object to delegate the work to an instance of these classes, instead of acting on its own.

Encapsulate varying behavior for the same routine based on an object’s state. This can be a cleaner way for an object to change its behavior at runtime without resorting to large monolithic conditional statements.

3.9 Strategy

PURPOSE

The Strategy's main goal is to define a family of algorithms that perform a similar operation, each with a different implementation.

The original object, called context, holds a reference to a strategy object and delegates it executing the behavior. In order to change the way the context performs its work, other objects may replace the currently linked strategy object with another one.

3.10 Template Method

PURPOSE

Is a behavioral design pattern that allows you to defines a skeleton of an algorithm in a base class and let subclasses override the steps without changing the overall algorithm's structure

The Template Method pattern is quite common in PHP frameworks. The pattern simplifies the extension of a default framework's behavior using the class inheritance..

3.11 Visitor

PURPOSE

Represents an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates

In other words the visitor pattern allows you to separate algorithms from the object Skelton on which it performs.
At its core it allows for adding functions to classes without modifying the classes themselves.

4.0 Repository

PURPOSE

Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects. Repository encapsulates the set of objects persisted in a data store and the operations performed over them

A repository is a separation between a domain and a persistent layer. The repository provides a collection interface to access data stored in a database, Data is returned in the form of objects.