Behavioral Design Patterns

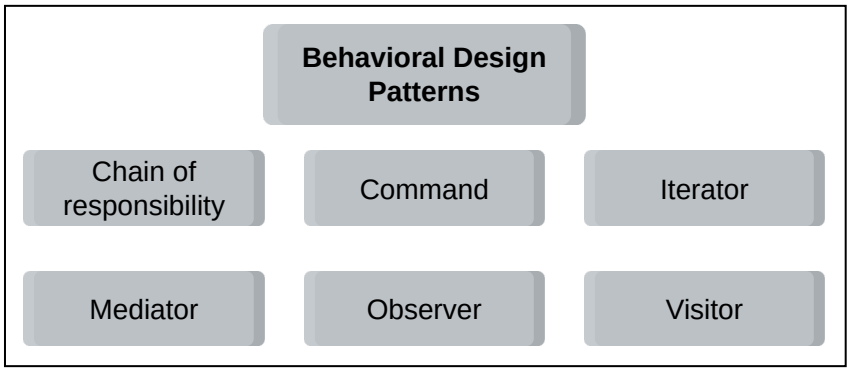
Get introduced to behavioral design patterns and learn to use them.

We'll cover the following

What are behavioral patterns?

These patterns ensure effective communication between different objects in a system, assign responsibilities to them, and make sure they all have synchronized information.

The chart below shows the patterns that fall under this category:



Behavioral design patterns

Chain of Responsibility pattern

The **Chain of Responsibility pattern** allows a request sent by a client to be received by more than one object. It creates a chain of loosely-coupled objects that, upon receiving the request, either handle it or pass it to the next handler object.

A common example of this pattern is event bubbling in DOM. An event propagates through different nested elements of the DOM until one of them handles it.

Command pattern

The **Command pattern** allows encapsulation of the requests or operations into separate objects. It decouples the objects that send requests from the objects responsible for executing those requests.

Consider an example where the client is accessing the methods of an API directly throughout the application. What will happen if the implementation of that API changes? The change will have to be made everywhere the API is being used. To avoid this, we could use abstraction and separate the objects requesting from those implementing the request. Therefore, if a change occurs, only the object making the call will need to change.

Iterator pattern

The **Iterator pattern** allows the definition of various types of iterators that can be used to sequentially iterate a collection of objects without exposing the underlying form.

Iterators encapsulate how the traversal occurs in an iteration. Most languages have built-in iterators such as IEnumerable and IEnumerator. Iterators follow the behavior where they call a `next` function and step through a set of values until they reach the end. To do this, they need to maintain a reference to the current position as well as the collection they are traversing. Hence, an iterator has functions such as `next`, `hasNext`, `currentItem`, and `each`.

Mediator pattern

It is a behavioral pattern that allows a mediator (a central authority) to act as the coordinator between different objects instead of the objects referring to each other directly. A mediator, as the name implies, is a central authority through which various components can communicate. It allows the loose coupling of objects.

A real-life example is a chat application. Here, the chat box acts as the mediator through which various users interact with one another.

Observer pattern

The **Observer pattern** is an important behavioral design pattern. It allows objects (observers) that have subscribed to an event to wait for input and react to it when notified. Therefore, they don't have to continuously keep checking whether the input has been provided or not. The main subject maintains a list of all the observers, and whenever the event occurs, it notifies the observers so they can update their states accordingly.

Let's look at a real-life example that we can map to this pattern. Consider a website that posts interesting articles. Every day, you visit the site to check for new articles, and if there are none, you revisit after some time. What if you get a subscription to the website instead? Once you have the subscription, you'll get notified every time a new article is posted. So now, instead of checking the site every few hours, you just wait for the notification about a new article.

Visitor pattern

The **Visitor pattern** allows the definition of new operations to the collection of objects without changing the structure of the objects themselves. This allows us to separate the class from the logic it implements.

The extra operations can be encapsulated in a visitor object. The objects can have a visit method that accepts the visitor object. The visitor can then make the required changes and perform the operations on the object that received it. This allows the developers to make future extensions, extend the libraries/frameworks, etc.

When to use behavioral design patterns?

Behavioral Design Patterns	When to use
Chain of Responsibility pattern	• It can be used where a program is written to handle various requ ways without knowing the sequence and type of requests before • It can be used in the process of *event bubbling* in the DOM, where propagates through the nested elements, one of which may choo event.
Command pattern	• It can be used to queue and execute requests at different times. • It can be used to perform operations such as “reset” or “undo”. • It can be used to keep a history of requests made.
Iterator pattern	• This pattern can be used when dealing with problems explicitly for designing flexible looping constructs and accessing elements collection without knowing the underlying representation. • It can be used to implement a generic iterator that traverses any independent of its type efficiently.
Mediator pattern	• It can be used to avoid the tight coupling of objects in a system w • It can be used to improve code readability. • It can be used to make code easier to maintain.
Observer pattern	• It can be used to improve code management by breaking down l into a system of loosely-coupled objects. • It can be used to improve communication between different par • It can be used to create a one-to-many dependency between obje coupled.
Visitor pattern	• It can be used to perform similar operations on different objects • It can be used to perform specific operations on different objects structure. • It can be used to add extensibility to libraries or frameworks.

Now that we have gone through all the design patterns, let's test the concepts we have learned so far in the next quiz lesson.

← Back

Structural Design Patt...

Next →

Quiz: Design Patterns

Mark as Completed