Behavioural Design Patterns

Contents

Overview

Behavioural design patterns are designed to improve and streamline interaction between objects.

The behavioural patterns are:

- Chain of Responsibility
- Command
- State
- Observer
- Strategy
- Interpreter
- Iterator
- Mediator
- Memento
- Null Object

Tutorial

Chain of Responsibility Pattern

Chain of Responsibility is a behavioural design pattern that lets you pass requests along a chain of handlers.

This pattern is made up of two main roles:

- Handler - defines an interface for handling requests, handles the request itself and implements the successor link.
- Client - initiates the request to a handler object in the chain.

Using the Chain of Responsibility pattern allows us to link these handlers into a chain. Each linked handler has a field for storing a reference to the next handler in the chain. In addition to processing a request, handlers pass the request further along the chain. The request travels along the chain until all handlers have had a chance to process it.

The handler can also stop, and therefore stop passing to other handlers down the chain, which stops any further processing.

Below is an example of how the Chain of Responsibility pattern can be used:

▶ Chain of Responsibility

- Use the Chain of Responsibility pattern when your program is expected to process different kinds of requests in various ways, but the exact types of requests and their sequences are unknown beforehand.

- Use the pattern when it's essential to execute several handlers in a particular order.

Chain of Responsibility Advantages

- You can control the order of request handling.
- Adheres to the Single Responsibility Principle. You can decouple classes that invoke operations from classes that perform operations.
- Adheres to the Open/Closed Principle. You can introduce new handlers into the app without breaking the existing client code.

Chain of Responsibility Disadvantages

- Some requests may end up unhandled

Strategy Pattern

The **Strategy** pattern allows for multiple implementations of an algorithm at runtime, depending on the specific implementation we want to use.

This can be useful for when we have several implementations of the same algorithm that are used in different cases but still want our code to be flexible enough to run any of those implementations on-the-fly.

For instance, let's say we have three ways of connecting a phone to a speaker:

▶ Three connections

At the moment, if we want to make multiple connections, or even switch from one connection method to another, we would need to manually create a new instance of one of these classes and use its individual `connectToX()` method every time.

Since each connection method broadly does the same sort of thing, we can simply extract these three different connection strategies out to an interface instead:

▶ Strategising