Chain Of Responsibility Design Pattern:

**What it does**

As the name suggests, the chain of responsibility pattern creates a chain of receiver objects for a request.  It is used when sender sends any request and it is not clear which receiver will handle this request. In this pattern, normally each receiver contains reference to another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

**Where to use**

1. When more than one object can handle a request and the handler is unknown.  
2. When the group of objects that can handle the request must be specified in dynamic way.  
3. ATM ( which receiver will handle the amount).  
4. Vendor Machine ( which kind of beverage will be floated).  
5. Logger System (which kind of log will be printed).  
6. When you want to decouple a request’s sender and receiver.  
7. Multiple objects, determined at runtime, are candidates to handle a request.  
8. When you don’t want to specify handlers explicitly in your code.  
9. When you want to issue a request to one of several objects without specifying the receiver explicitly.

**Steps**

1. Abstract Chain of responsibility class: There will be static variables for each receivers. The constructor will take the next receiver as variable. And it has one function and if the next receiver is not null it will call the same function of the next receiver.  
2. Concrete chain of responsibility classes/ different handlers/processors/receivers/controllers: They will extend the abstract chain of responsibility class. If the state match will the current class it will execute the function otherwise it will call the same function of its parent.  
3. Client code: Chains or steps will be defined here. A object will be created using abstract chain of responsibility class and concrete chain of responsibility classes like this format AbstractChainOfResponsibility obj = new ConcreteChainOfResponsibility1( new ConcreteChainOfResponsibility2 ( new ConcreteChainOfResponsibility3( null ) ) );  
Finally call the method. It will first call ConcreteChainOfResponsibility1 then ConcreteChainOfResponsibility2 and then   
ConcreteChainOfResponsibility3.

**Special cases (if any)**

placeholder

**Advantages**

1. It promotes loose coupling between senders and receivers of requests, as each handler in the chain only knows about its immediate successor.  
2. It supports adding new handlers or modifying the chain structure easily, making it scalable to accommodate changes in requirements.  
3. The pattern simplifies the handling of complex or conditional logic by breaking it down into smaller, manageable components.  
4. If a handler fails to process a request, it can pass the request along the chain to the next handler, ensuring fault tolerance and graceful degradation.  
5. New handlers can be added to the chain or existing handlers can be modified without affecting the client code, making the pattern highly extensible.  
6. The chain structure and handlers' order can be configured dynamically at runtime, allowing for adaptive behavior based on changing requirements.  
7. Each handler is responsible for a specific task or type of request, adhering to the SRP (**Single Responsibility Principle)** and making the codebase easier to maintain.  
8. Handlers can be tested independently using unit tests, facilitating the testing of individual components.  
9. The order in which handlers are added to the chain determines the sequence in which they process requests, enforcing **order of execution.**

**Disadvantages**

1. Implementing and managing a chain of handlers can introduce complexity, especially in systems with many handlers.  
2. Each request must traverse the entire chain until a suitable handler is found, potentially leading to performance overhead, especially in long chains.  
3. Determining which handler in the chain is responsible for handling a particular request can be challenging, making debugging more difficult.  
4. If no handler in the chain can handle a specific request, it may go unprocessed, leading to potential issues or unexpected behavior.  
5. Understanding the flow of requests through the chain and identifying the responsibilities of each handler may be challenging for developers, especially in large or complex chains.  
6. Ensuring that the chain remains intact and that all requests are processed correctly may require additional effort and validation, especially in systems with multiple entry points or complex request flows.  
7. Testing the entire chain can be tough, needing lots of integration testing.

**Code**

Coding Concepts

**Difference with similar pattern**

Decorator Design Pattern: Both patterns involve a layered structure where objects can be wrapped or linked to achieve the desired functionality. Chain of responsibility design pattern passes a request through a chain of handlers until one handles it. Decorator design pattern dynamically adds new features or behaviors to objects.  
May be state and command design pattern but [[later]]

**Diagram**

Tutorialspoint