# Chain of Responsibility [GoF]

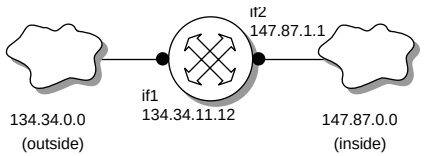
## Intent

Sending requests along a chain of objects each having a chance to handle the request. Avoid coupling the sender of a request with the concrete receiver.

## Motivation

Suppose a router having the duty of forwarding packets from one network to the other. However, the router must refrain from forwarding of packets meeting specific criteria in terms of filter lists.

A sketch of a simple network might look like:





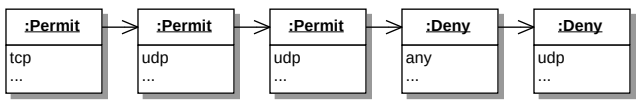



An imaginary filter list associated to interface `if1` might look like:

```
// p/d   proto src host/nw src prt  dst host/nw    dst prt
deny     any   123.123.1.1 0        147.87.0.0     0
permit   tcp   0.0.0.0     0        147.87.64.7    80
permit   udp   0.0.0.0     0        147.87.1.1     53
permit   tcp   0.0.0.0     0        147.87.62.5    21
deny     udp   0.0.0.0     0        147.87.0.0     0
```

Upon the receipt of a packet at interface `if1`, the interface of the router first checks its associated filter list. If the filter list decides that the packet can be routed, then the interface passes the packet to the forwarder.

Suppose that each entry is associated with a `FilterEntry` object. Then we can arrange the set of entries in a corresponding chain of `FilterEntry` objects:







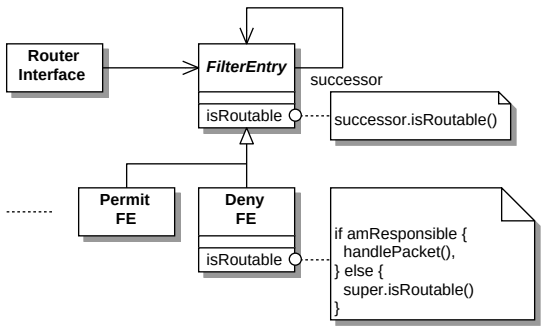

Then, each `FilterEntry` object does the following thing:

1. It first checks whether it is responsible for the request.
   For example, if a filter entry object for handling TCP packets actually receives a TCP packet then it feels responsible for it. If, however, the same filer entry object receives a UDP packet then it does not feel responsible for it; it the passes the packet to the next filter entry object in the chain.
2. It the checks whether it can handle the request.
   For example, if the filter entry object is for TCP packets, and it actually receives a TCP packet, it checks if the source/destination address and port informations matches the criteria of the filter entry object. If the criteria match then the filter entry object reacts accordingly. If it does not match then it passes the packet to the next filter entry object in the chain.
3. In all other cases, it forwards the packet to the next filter entry object in the chain.
4. If a filter entry object is the last instance in the chain of filter entry objects, and neither of the above cases do apply, then it denies the routing of the packet.





To ensure the integrity of the chain of successors, you may introduce a base class for every concrete filter entry subclass:



## Applicability

Use Chain of Responsibility when

- more than one object may handle the request;
- the handler is not know *a priori*;
- you want to send a request to one of several objects without specifying the receiver explicitly;
- the set of objects that can handle a request should be specified dynamically.





## Participants

- **`Handler`** (`FilterEntry`)
  - defines an interface for handling requests
  - implements the successor link (optional).
- **`ConcreteHandler`** (`PermitFilterEntry`, `DenyFilterEntry`)
  - handles request it is responsible for
  - if it can handle the request, it does so; otherwise it forwards the request to its successor.

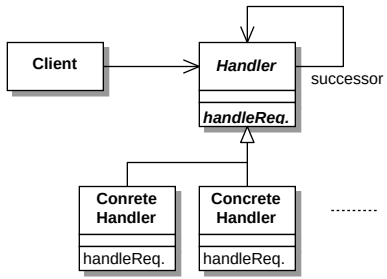
## Collaborations

- When a client issues a request, the request propagates along the chain until a concrete handler object takes responsibility to handle it.





## Structure








## Consequences

Benefits and liabilities:

1. *Reduced coupling.* The pattern frees the sender from knowing which other object handles a request.
2. *Added flexibility in assigning responsibilities to object.* The pattern gives you flexibility in distributing responsibilities among objects. You can even change the chain of responsible objects at run-time.
3. *Receipt isn't guaranteed.* Since the request has no explicit receiver, there is no guarantee that it will be handled — a request can fall off the end of the chain. A request can also go unhandled if the chain is not configured properly.

## Implementation

1. *Connecting successors.* If there are no preexisting references for defining a chain (e.g., via a Composite), then you'll have to introduce them yourself. See the code section to see how this can be done in the base class.
2. *Representing requests.* Possibilities are:
   – hard-coded method invocations (as in our motivation example). You can forward only a fixed set of requests that the Handler class defines.
   – single handler method with request code. This supports an open-ended set of requests. Sender and receiver must agree how the request should be encoded.





A concrete subclass of class `FilterEntry` is given next. Let's consider the `PermitFilterEntry` class:

```
public class PermitEntry extends FilterEntry
{
   private Pattern
      sourceAddressPattern,
      ...;

   private String protocolType; // one of {tcp, udp, any}

   ...
   // Class content continued on next slide...
}
```





## Sample Code

Let's consider a sketch of the code of the example in the Motivation section.

The base class for `FilterEntry` objects might look like:

```
public abstract class FilterEntry {
   private FilterEntry successor;

   public FilterEntry(FilterEntry aSuccessor) {
      successor = aSuccessor;
   }

   public boolean isRoutable(Packet packet) {
      if (successor != null) {
         return successor.isRoutable(packet);
      } else {
         return false;  // depends on the semantics of the
                        // filter chaine
      }
   }
} // FilterEntry
```





The constructor of the `PermitEntry` class takes a `FilterEntry` as parameter. This value is used by the superclass to initialize the successor.

```
public PermitEntry(FilterEntry aSuccessor,
          String aSourceAddressPatternString,
          String aSourcePortPatternString,
          String aProtocolType, // one of {tcp, udp, any}
          ...)
{
   super(aSuccessor);
   sourceAddressPattern =
      Pattern.compile(aSourceAddressPatternString);
   sourcePortPattern =
      Pattern.compile(aSourcePortPatternString);
   protocolType = aProtocolType;
   ...
}
```

The rest of the `PermitEntry` class is the handler method `isRoutable` as well as any necessary private helper methods:

```
public boolean isRoutable(Packet packet)
{
   if (areResponsible(packet)) {
      boolean rval;
      // compute rval if responsible
      return rval;
   } else {
      return super.isRoutable(packet);
   }
}

private boolean areResponsible(Packet aPacket)
{
   return
      protocolType.equals(aPacket.getProtocolType())
      ||
      protocolType.equals("any");
}
```





A private helper method for this example might look like:

```
// Helper methods that check individual criterie.
private boolean permitSourceAddress(Packet packet)
{
   Matcher m =
    sourceAddressPattern.matcher(packet.getSourceAddress());
   return m.matches();
}

// Remaining helpers omitted...
```

## Related Patterns

- Often applied in conjunction with Composite. There, a component's parent can act as its successor.