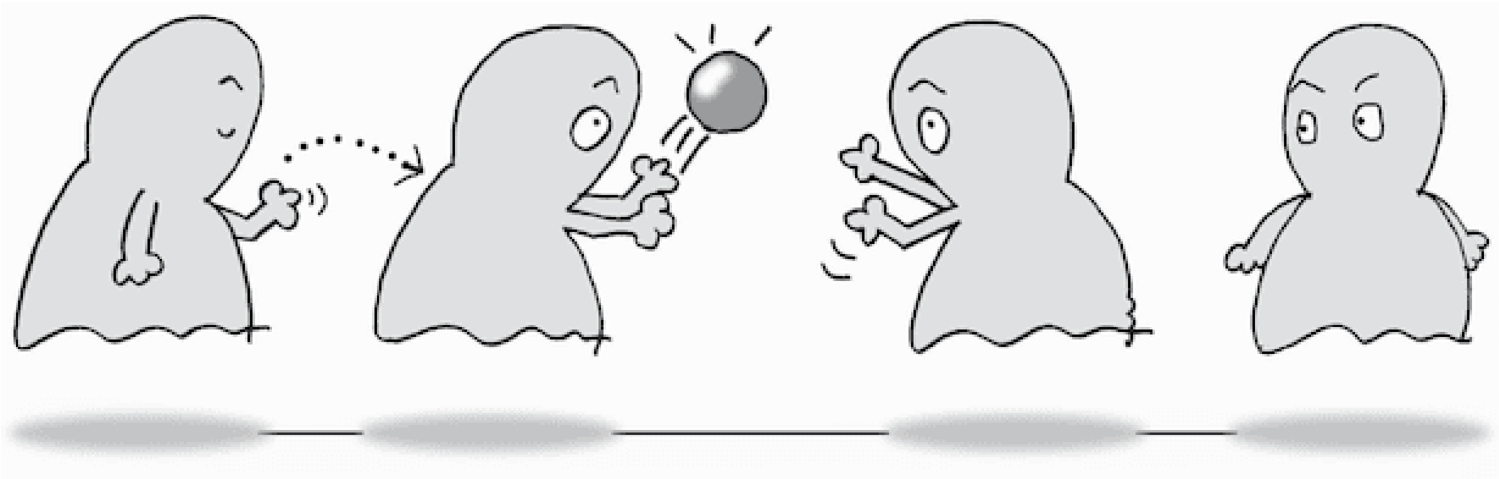
Chain of Responsibility Pattern

Pass Requests Along Chain of Handlers



Chain of Responsibility Pattern

Think of **escalation procedures** in organizations:

- **Customer service**: First-level support handles simple issues
- **Technical team**: Second level handles complex problems
- **Manager**: Third level handles complaints and escalations
- **Director**: Final level handles critical issues

Each **handler** tries to resolve the issue; if they can't, they **pass it up** the chain.

The Problem

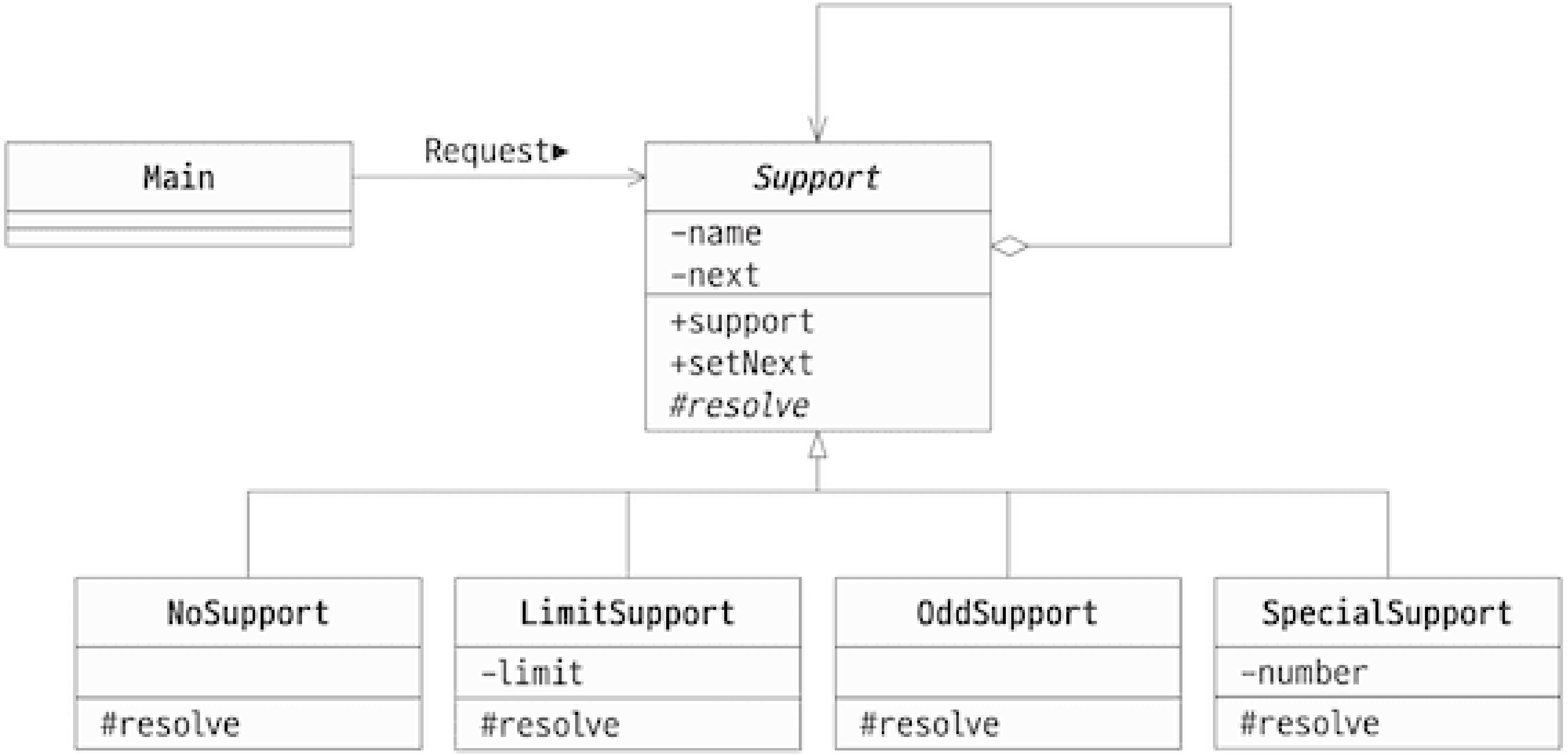
- We have a **support system** with multiple **handlers** (L1 → L2 → L3 → Specialist).
- **Different issues** require **different handlers**.
- At the time of sending a request, we **don't know** which handler can solve it.
- We want to **avoid coupling** the sender to any specific handler and allow **flexible reordering**/extension of handlers.

Challenge: Route a request through a **chain** of handlers so that **the first capable handler** processes it; otherwise, pass it along—without the sender knowing who will handle it.

The *Chain of Responsibility* as the Solution

- We create a **chain** of potential *handlers*.
- Each **handler** has a chance to process the request.
- If a **handler** can't process it, the request moves to the **next handler** in the chain.

The Solution (Design)



Step 1: Understand the Players

In this design, we have players:

- *Handler* (Support)
  - **ConcreteHandler** (NoSupport, LimitSupport, SpecialSupport, OddSupport)
- *Client* - sends requests to the chain

Step 2: Chain Structure

- **Handlers** maintain a reference to the next handler in the chain.
- Each **handler** decides: handle request or pass to next handler.

Step 3: Understand abstractions (Handler)

- We have a *Handler* that `defines` the interface for handling requests and managing the chain.
    - *Handler* (Support) - maintains next handler reference and chain logic
    - **ConcreteHandler** - implements specific handling logic
- **Handlers** can be linked together in any order.

- Notice that **Handler** has a `set_next()` method to build the chain.
  - It also has a `support()` method that implements the chain logic.
- Notice that **ConcreteHandlers** implement `resolve()` to decide if they can handle the request.
  - Each handler has different criteria for what they can handle.

Step 4: Understand concretion (Handler)

- We have **NoSupport**, **LimitSupport**, **SpecialSupport**, **OddSupport** (handlers).
  - **NoSupport**: Never handles anything (always passes along)
  - **LimitSupport**: Handles troubles below a certain limit
  - **SpecialSupport**: Handles specific trouble numbers
  - **OddSupport**: Handles odd-numbered troubles

Code

- Main Method
- Handler Classes
- Request Processing

Main Method

```python
from trouble import Trouble
from no_support import NoSupport
from limit_support import LimitSupport
from special_support import SpecialSupport
from odd_support import OddSupport

def main():
    print("=== Chain of Responsibility Example ===\n")

    # Create handlers
    alice = NoSupport("Alice")
    bob = LimitSupport("Bob", 100)
    charlie = SpecialSupport("Charlie", 429)
    diana = OddSupport("Diana")

    # Build chain
    alice.set_next(bob).set_next(charlie).set_next(diana)

    # Send requests through chain
    for num in [33, 99, 150, 429, 500]:
        alice.support(Trouble(num))
```

Step 1: Create handlers with specific capabilities

```
alice = NoSupport("Alice")        # Never handles
bob = LimitSupport("Bob", 100)    # Handles < 100
charlie = SpecialSupport("Charlie", 429)  # Only handles 429
diana = OddSupport("Diana")       # Handles odd numbers
```

- Each **handler** has different *capabilities* and *responsibilities*.
- **Handlers** implement `resolve()` method with their specific logic.

Step 2: Build the chain

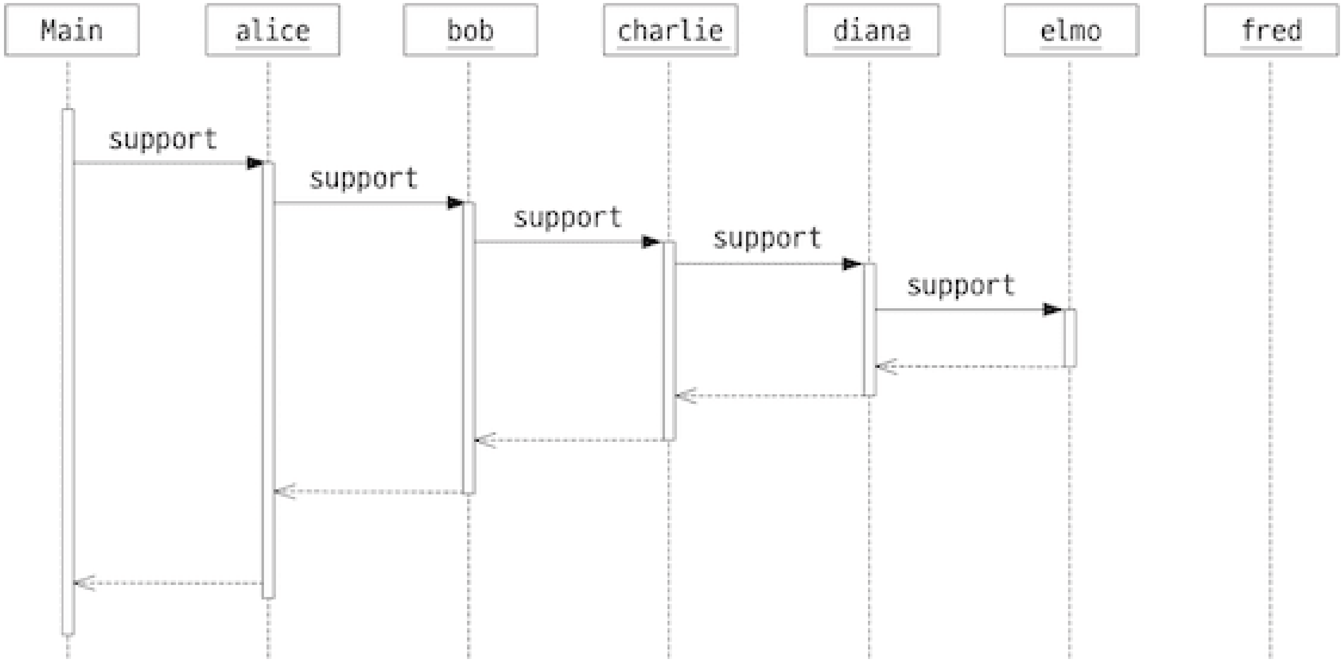
```
alice.set_next(bob).set_next(charlie).set_next(diana)
```

- **Chain**: Alice → Bob → Charlie → Diana
- Each **handler** knows the next handler in the sequence.
- Requests flow through this predetermined path.

Step 3: Send requests through the chain

```
alice.support(Trouble(33))   # Bob handles (< 100)
alice.support(Trouble(150))  # Diana handles (odd)
alice.support(Trouble(429))  # Charlie handles (special)
alice.support(Trouble(500))  # No one handles
```

- **Requests** automatically find the appropriate handler.
- If no handler can process the request, it's **rejected**.


Main
alice
bob
charlie
diana
elmo
fred
support
support
support
support
support

Handler Classes

```python
# support.py
from abc import ABC, abstractmethod

class Support(ABC):
    def __init__(self, name):
        self.name = name
        self.next = None

    def set_next(self, next_support):
        self.next = next_support
        return next_support

    def support(self, trouble):
        if self.resolve(trouble):
            self.done(trouble)
        elif self.next is not None:
            self.next.support(trouble)
        else:
            self.fail(trouble)

    @abstractmethod
    def resolve(self, trouble):
        pass
```

Concrete Handlers

```python
# limit_support.py
class LimitSupport(Support):
    def __init__(self, name, limit):
        super().__init__(name)
        self.limit = limit

    def resolve(self, trouble):
        return trouble.get_number() < self.limit

# special_support.py
class SpecialSupport(Support):
    def __init__(self, name, number):
        super().__init__(name)
        self.number = number

    def resolve(self, trouble):
        return trouble.get_number() == self.number

# odd_support.py
class OddSupport(Support):
    def resolve(self, trouble):
        return trouble.get_number() % 2 == 1
```

Discussion

Chain Processing Logic

```
def support(self, trouble):
    if self.resolve(trouble):
        self.done(trouble)           # I can handle it
    elif self.next is not None:
        self.next.support(trouble)   # Pass to next handler
    else:
        self.fail(trouble)           # No one can handle it
```

- Each **handler** first tries to resolve the request.
- If unsuccessful and the next handler exists, **passes the request along**.
- If no next handler, request **fails**.

Key Benefits

1. **Decoupling**: Sender doesn't know which handler processes the request
2. **Flexibility**: Add/remove/reorder handlers dynamically
3. **Single Responsibility**: Each handler focuses on specific criteria
4. **Open/Closed**: Add new handler types without changing existing code

Key Drawbacks

1. **No guarantee**: Request might not be handled by anyone
2. **Performance**: May traverse the entire chain before finding the handler
3. **Debugging**: Hard to trace which handler will process the request
4. **Chain management**: Need to carefully manage chain structure

When to Use Chain of Responsibility

- When **more than one object** can handle a request
- When you want to issue requests **without knowing** the receiver
- When **handler set** should be specified dynamically
- When you want to **avoid if-else chains** for request routing

When NOT to Use Chain of Responsibility

- When you have **only one handler** for each request type
- When **performance** is critical (chain traversal overhead)
- When **order matters** and must be guaranteed
- When handlers have **complex interdependencies**

Real-World Examples

- **Exception handling**: try-catch blocks with different exception types
- **Event handling**: GUI events bubbling up through the widget hierarchy
- **Authentication**: Multiple authentication methods (token, session, basic)
- **Logging**: Different log levels processed by different handlers

Related Patterns

- **Composite**: Chain often used in Composite structures for tree traversal
- **Decorator**: Both use recursive composition, but for different purposes
- **Template Method**: Handlers might use Template Method for processing steps

Chain vs Decorator

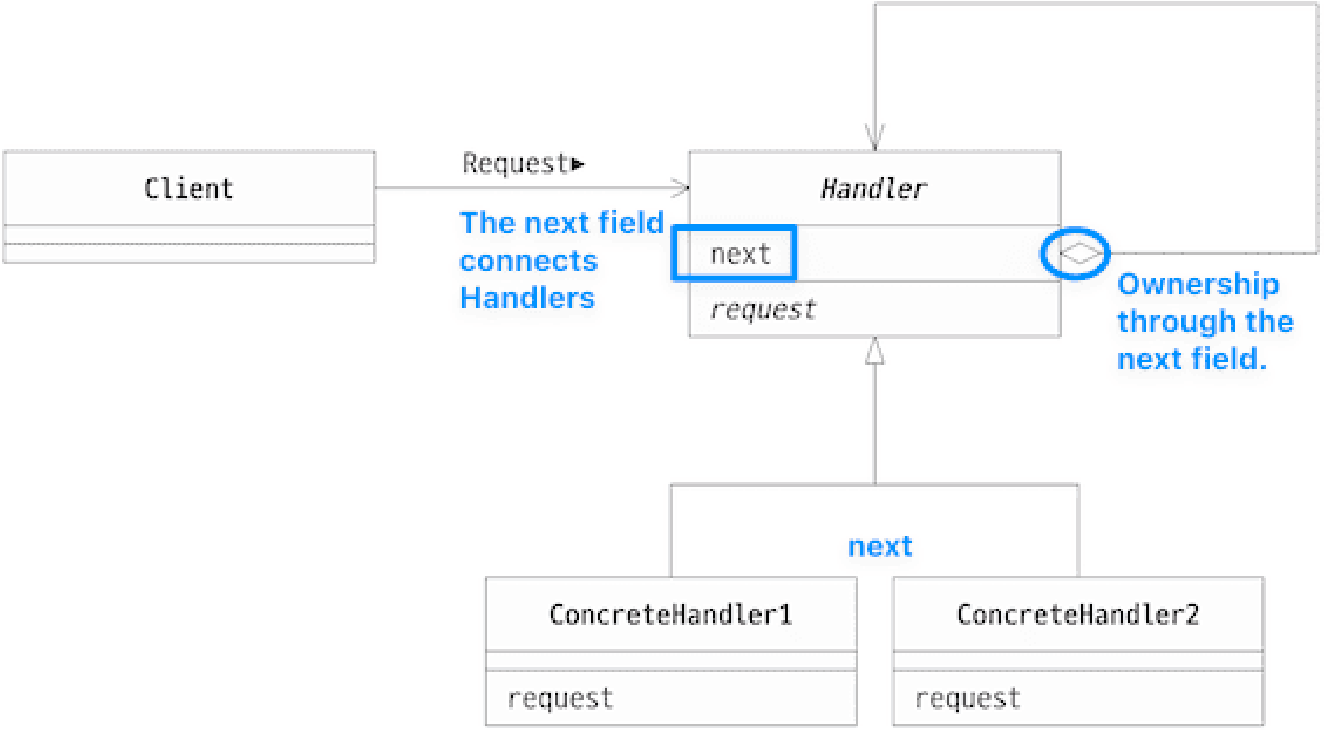
Chain of Responsibility:

- **One** handler processes the request
- Request **stops** when handled
- **Linear** processing path

Decorator:

- **All** decorators process the request
- Request **passes through** all decorators
- **Nested** processing (wrapping)

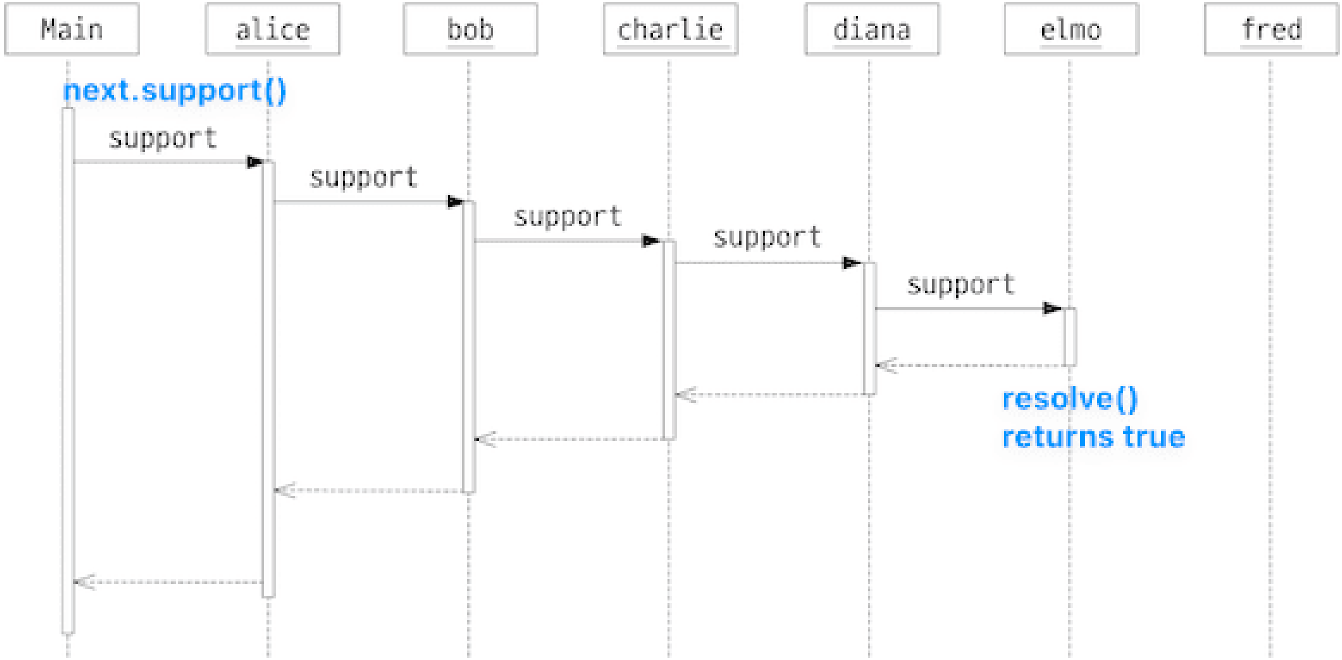
UML



The next field connects Handlers

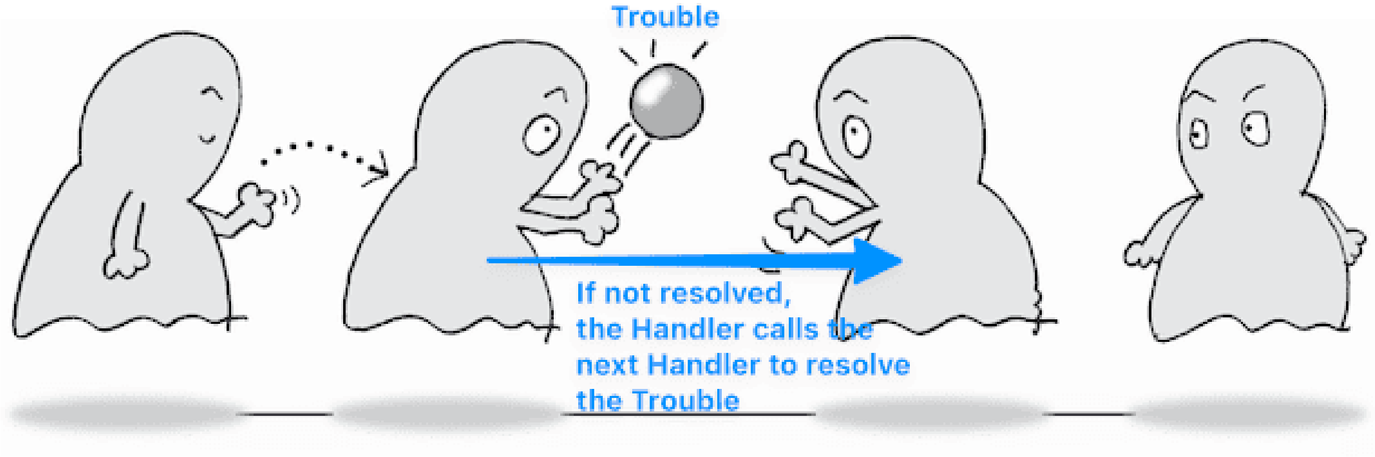
Ownership through the next field.

next



next.support()

resolve()
returns true


Trouble
If not resolved,
the Handler calls the
next Handler to resolve
the Trouble