

# CHAIN OF RESPONSIBILITY

Behavioral Pattern

# INTENT

Avoid coupling the sender of a request to its receiver by giving more than one object chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

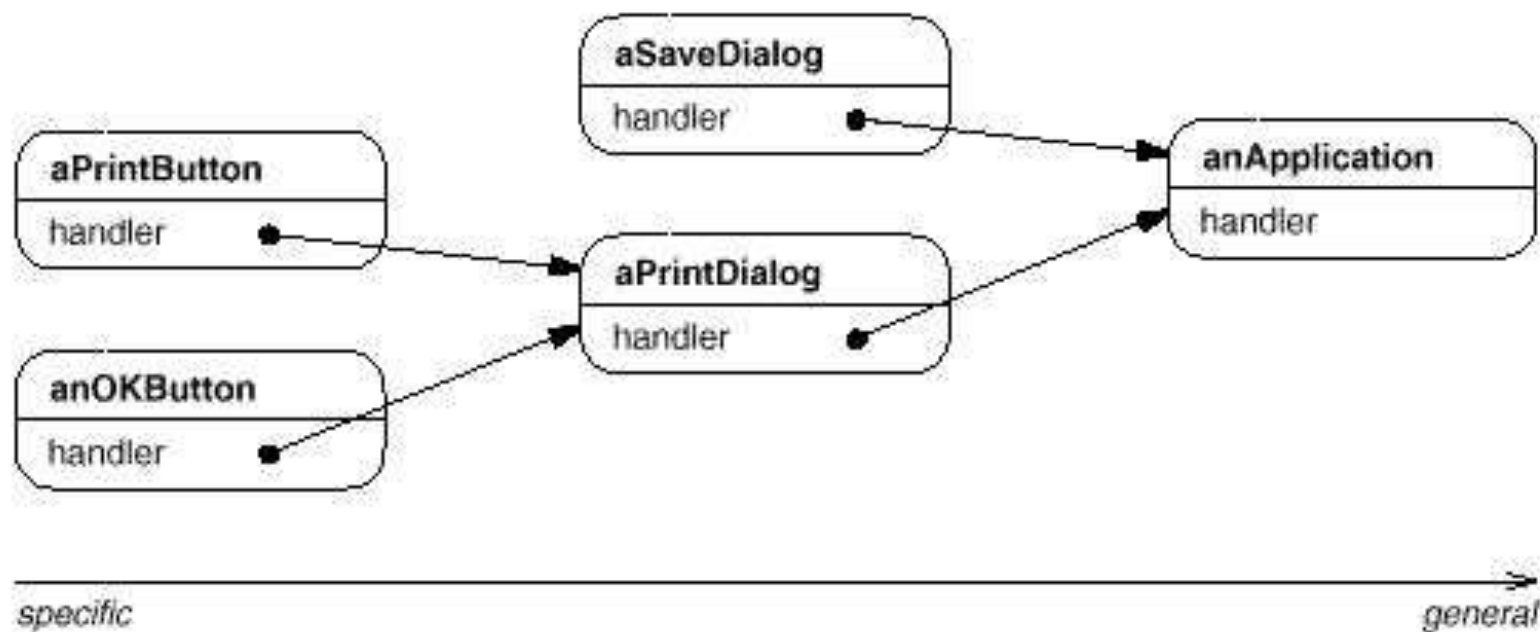
# MOTIVATION

Consider a context-sensitive help facility for a graphical user interface.

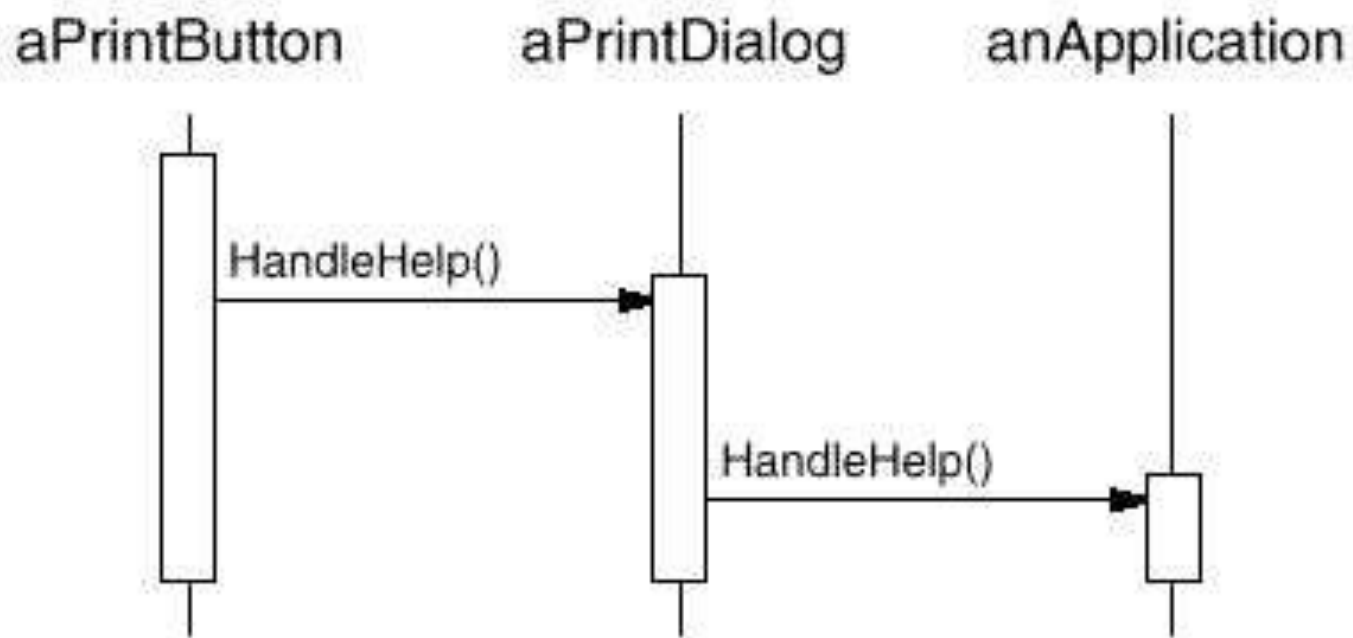
- ◉ If there is help for the button => show it, otherwise
- ◉ If there is help for the dialog => show it, otherwise
- ◉ If there is help for the application => show it, otherwise ...

*The problem: the object that ultimately provides the help isn't known explicitly to the object (e.g., the button) that initiates the help request.*

# MOTIVATION



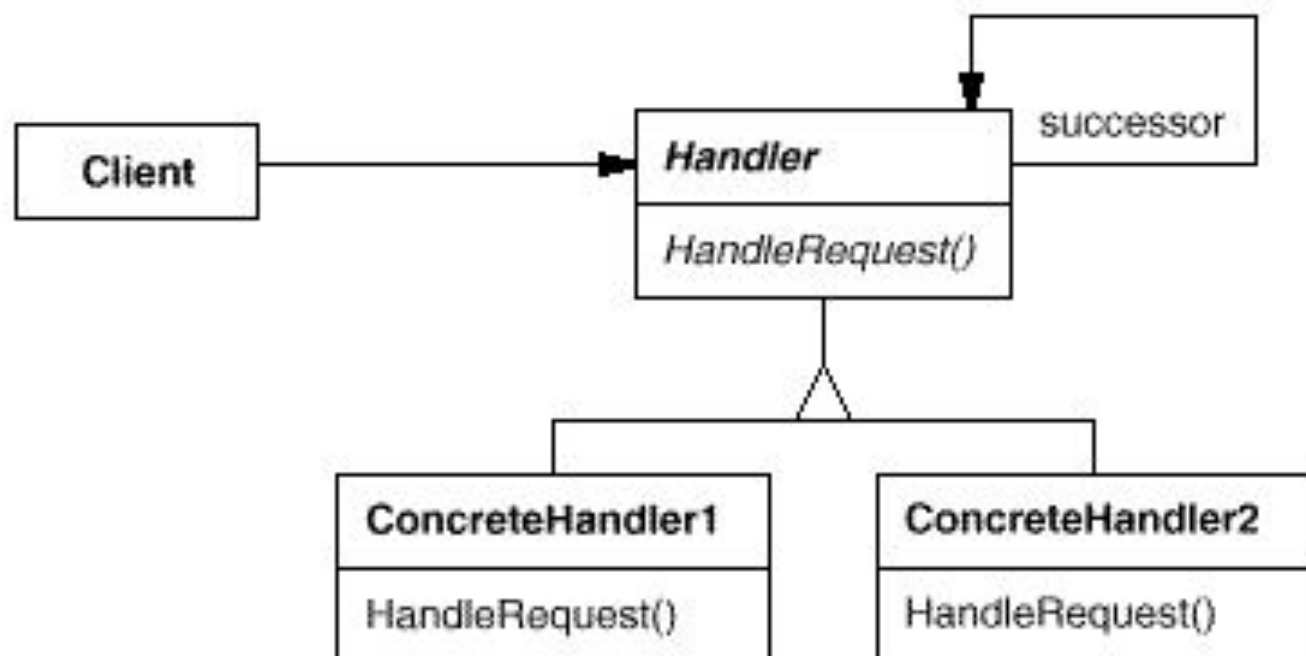
# MOTIVATION



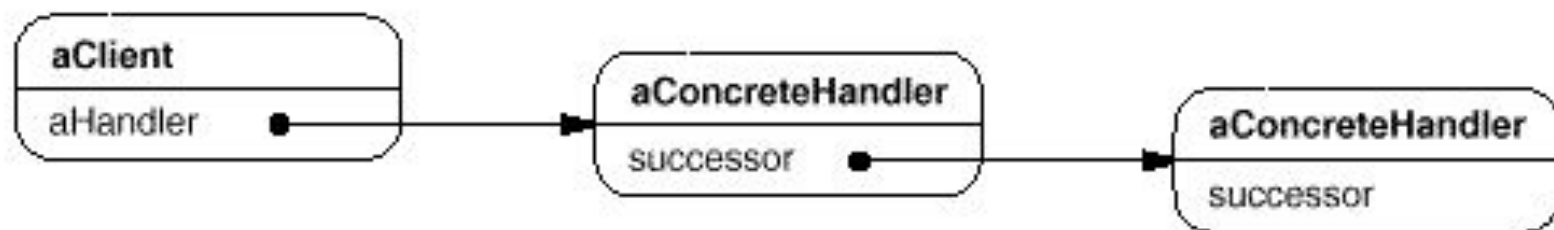
# APPLICABILITY

- ◉ More than one object may handle a request, and the handler isn't known *a priori*. *The handler should be ascertained automatically.*
- ◉ You want to issue 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.

# STRUCTURE



# OBJECT STRUCTURE





# PARTICIPANTS

- ◉ **Handler (HelpHandler)** - defines an interface for handling requests.
- ◉ **ConcreteHandler (PrintButton, PrintDialog)**
  - handles requests it is responsible for. It can access its successor. If the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.
- ◉ **Client** - initiates the request to a ConcreteHandler object on the chain.

# BENEFITS

- ◉ ***Reduced coupling:*** The pattern frees an object from knowing which other object handles a request. An object only has to know that a request will be handled "appropriately." Both the receiver and the sender have no explicit knowledge of each other, and an object in the chain doesn't have to know about the chain's structure.

# BENEFITS

- ◉ ***Added flexibility in assigning responsibilities to objects.*** Chain of Responsibility gives you added flexibility in distributing responsibilities among objects.

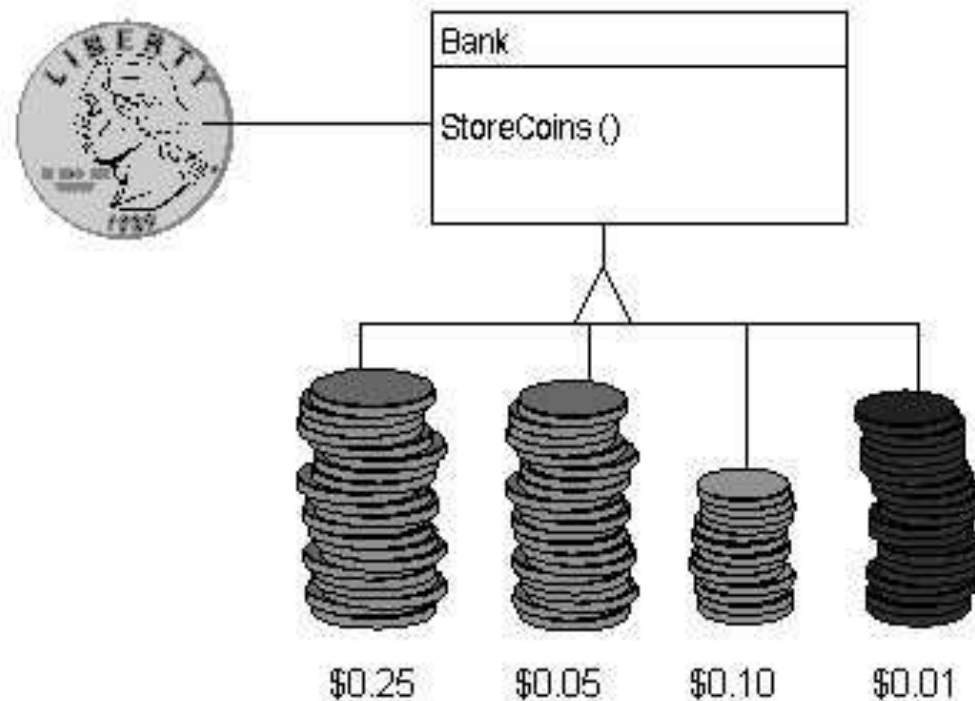
# DRAWBACKS

- ◉ ***Receipt isn't guaranteed***: Since a request has no explicit receiver, there's no guarantee it'll be handled—the request can fall off the end of the chain without ever being handled.

## RELATED PATTERNS

- ◉ Chain of Responsibility is often applied in conjunction with **Composite**, where a component's parent can act as its successor.

# NON-SOFTWARE EXAMPLE



A mechanical sorting bank uses a single slot for all coins. As each coin is dropped, a *Chain of Responsibility* determines which tube accommodates each coin. If a tube cannot accommodate the coin, the coin is passed on until a tube can accept the coin.

# EXAMPLE 1

- Scenario: The designers of a set of GUI classes need to have a way to propagate GUI events, such as `MOUSE_CLICKED`, to the individual component where the event occurred and then to the object or objects that are going to handle the event
- Solution: Use the Chain of Responsibility pattern. First post the event to the component where the event occurred. That component can handle the event or post the event to its container component (or both!). The next component in the chain can again either handle the event or pass it up the component containment hierarchy until the event is handled.
- This technique was actually used in the Java 1.0 AWT

# EXAMPLE 1

```
public boolean action(Event event, Object obj) {  
    if (event.target == test_btn)  
        doTestBtnAction();  
    else if (event.target == exit_btn)  
        doExitBtnAction();  
    else  
        return super.action(event,obj);  
    return true;    // Return true to indicate the event has been  
                    // handled and should not be propagated further.  
}
```



# EXAMPLE 1

- In Java 1.1 the AWT event model was changed from the Chain of Responsibility (CoR) pattern to the Observer pattern. Why?
  - ⇒ Efficiency: GUIs frequently generate many events, such as `MOUSE_MOVE` events. In many cases, the application did not care about these events. Yet, using the CoR pattern, the GUI framework would propagate the event up the containment hierarchy until some component handled it. This caused the GUI to slow noticeably.
  - ⇒ Flexibility: The CoR pattern assumes a common Handler superclass or interface for all objects which can handle chained requests. In the case of the Java 1.0 AWT, every object that could handle an event had to be a subclass of the `Component` class. Thus, events could not be handled by non-GUI objects, limiting the flexibility of the program.

## EXAMPLE 1

- The Java 1.1 AWT event model uses the Observer pattern. Any object that wants to handle an event registers as an event listener with a component. When an event occurs, it is posted to the component. The component then dispatches the event to any of its listeners. If the component has no listeners, the event is discarded. For this application, the Observer pattern is more efficient and more flexible!

## EXAMPLE 2

- Scenario: We are designing the software for a security monitoring system. The system uses various sensors (smoke, fire, motion, etc.) which transmit their status to a central computer. We decide to instantiate a sensor object for each physical sensor. Each sensor object knows when the value it is sensing exceeds some threshold(s). When this happens, the sensor object should take some action. But the action that should be taken may depend on factors other than just the sensor's value. For example, the temperature in a warehouse exceeding 100 °F may require a totally different action than if the sensor is in a freezer. We want a very scalable solution in which we can use our physical sensors and sensor objects in any environment. What can be do??

## EXAMPLE 2

- Solution: Use the Chain of Responsibility pattern. Aggregate sensor objects in a containment hierarchy that mirrors the required physical zones (areas) of security. The alarm generated by a sensor is passed up this hierarchy until some zone handles it.

## EXAMPLE 3

```
public class Processor extends AChain {
    private static java.util.Random rn =
        new java.util.Random();
    private String id;

    public Processor (String tmpid){
        id =tmpid;
    }
    public void sendToChain(String task){
        if (rn.nextInt(2) == 0) {
            System.out.println
                ( "Processor " + id + " executes " + task);
        }
        else{
            System.out.println
                ( "Processor " + id + " is busy ");
            if (nextChain != null)
                nextChain.sendToChain(task);
        }
    }
}
```

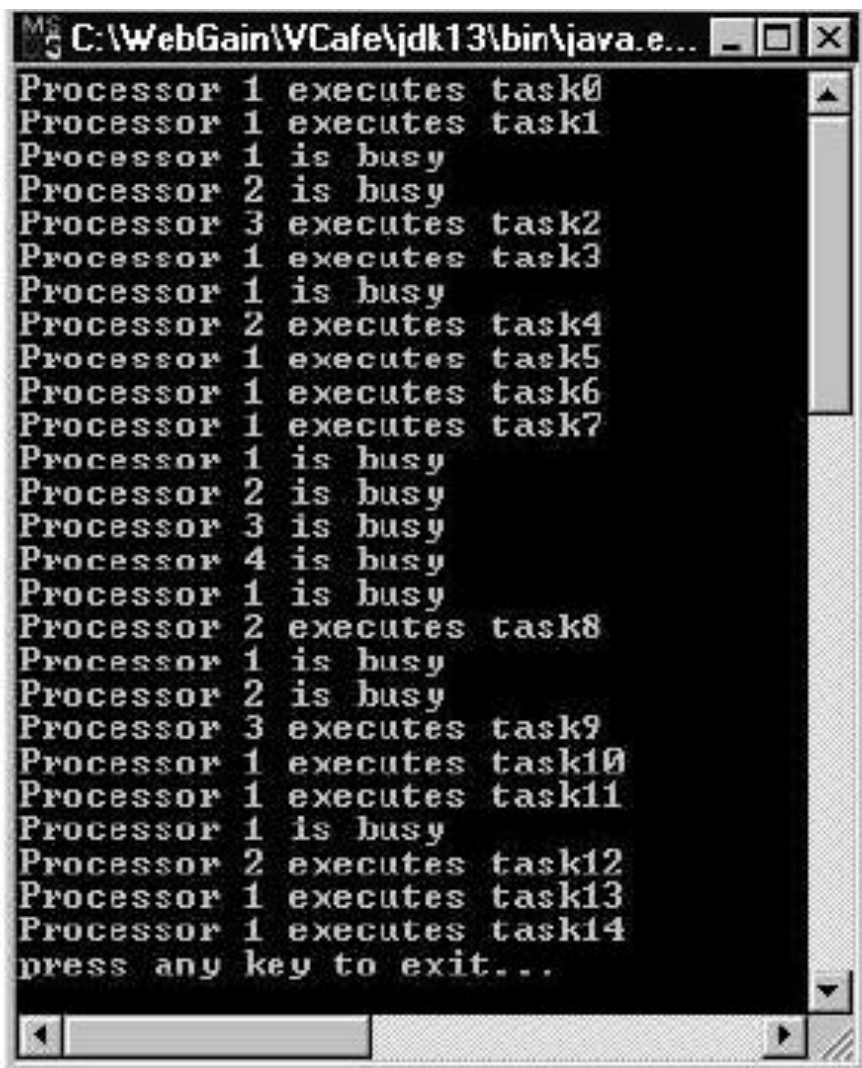
```
public interface Chain
{
    public void addChain(Chain c);
    public void sendToChain(String task);
    public Chain getChain();
}

public abstract class AChain implements Chain
{
    protected Chain nextChain;
    public void addChain(Chain c){
        nextChain = c;
    }
    public Chain getChain(){
        return nextChain;
    }
    public abstract void sendToChain(String task);
}
```



## EXAMPLE 3

```
public class Application {  
    static public void main(String args[]) {  
        String task="";  
        Processor p1= new Processor("1");  
        Processor p2= new Processor("2");  
        Processor p3= new Processor("3");  
        Processor p4= new Processor("4");  
        p1.addChain(p2);  
        p2.addChain(p3);  
        p3.addChain(p4);  
        p4.addChain(p1);  
        for (int i=0; i<15; i++){  
            task="task"+String.valueOf(i);  
            p1.sendToChain(task);  
        }  
    }  
}
```



The screenshot shows a Java application window titled "C:\WebGain\VCafe\jdk13\bin\java.e...". The window contains a text area with the following output:

```
Processor 1 executes task0  
Processor 1 executes task1  
Processor 1 is busy  
Processor 2 is busy  
Processor 3 executes task2  
Processor 1 executes task3  
Processor 1 is busy  
Processor 2 executes task4  
Processor 1 executes task5  
Processor 1 executes task6  
Processor 1 executes task7  
Processor 1 is busy  
Processor 2 is busy  
Processor 3 is busy  
Processor 4 is busy  
Processor 1 is busy  
Processor 2 executes task8  
Processor 1 is busy  
Processor 2 is busy  
Processor 3 executes task9  
Processor 1 executes task10  
Processor 1 executes task11  
Processor 1 is busy  
Processor 2 executes task12  
Processor 1 executes task13  
Processor 1 executes task14  
press any key to exit...
```