

CHAPTER 8

Behavioral Patterns: Chain of Responsibility and Command

The two behavioral patterns we will explore in this chapter—Chain of Responsibility and Command—are concerned with passing requests for action to appropriate objects. The respective patterns choose the objects to pass the requests to in different ways.

Chain of Responsibility Pattern

Role

The Chain of Responsibility pattern works with a list of `Handler` objects that have limitations on the nature of the requests they can deal with. If an object cannot handle a request, it passes it on to the next object in the chain. At the end of the chain, there can be either default or exceptional behavior.

Illustration

The Chain of Responsibility pattern is encountered frequently in real life. Imagine members of the public (such as you and me) approaching a bank, a government office, a hospital, or a similar institution with a request. In the case of a bank, we know that a first level of clerks will handle straightforward requests. Those requests requiring more inspection will be sent up to a supervisor, and ultimately the manager might have to authorize or rule on a request. The clerk, supervisor, and manager together are an example of a chain of responsibility. Of course, sometimes clerks can deal with matters quite well on their own, as illustrated in Figure 8-1.*

Within the bank illustration, we can immediately see that it is the levels that form the chain (clerk, supervisor, manager), and there could well be more than one individual at each level. In human terms, such chains do not generally get very long (or else

* From S. Francis, H. Dugmore, and Rico, *www.madamandev.co.za*, printed with permission.

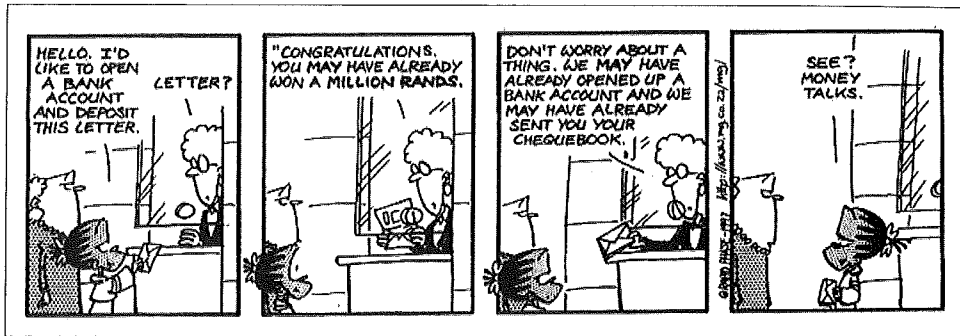


Figure 8-1. Chain of Responsibility pattern illustration—a bank clerk deals with a request

customers would get very frustrated), but theoretically there is no limit. Typically, clerks will stagger their tea and lunch breaks so that there is always someone on duty; they join and leave the chain dynamically, and at different times.

Design

Imagine a client that has a request for a handler. The Chain of Responsibility pattern distances the handler from the client by passing the request along a hidden chain of handlers until one that will handle it is found.

To handle a request, the handler must have the appropriate resources, knowledge, and permissions. The handlers themselves decide whether to handle a request and can do so on the basis of workload, resource limitations, policy issues, or any other good reason.

The Chain of Responsibility pattern is a simple pattern in that it consists of a single, uniform interface that is implemented by multiple handlers (as many as are necessary or available). Each handler is linked in some way to a successor so that requests can be passed on if necessary. Depending on the language, the link to the successor is either internal or external to the handler itself. The UML for the Chain of Responsibility pattern is shown in Figure 8-2.

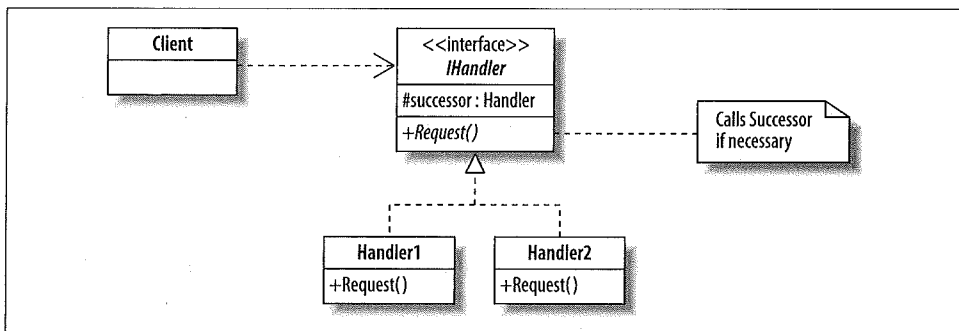


Figure 8-2. Chain of Responsibility pattern UML diagram

The roles assumed by the players in the pattern are as follows:

Client

A class that initiates a Request

IHandler

The interface for different Handlers

Handler1, Handler2, *etc.*

The classes that are available as Handlers

Successor

The link to the next Handler object

Request

The request from the Client

QUIZ

Match the Chain of Responsibility Pattern Players with the Bank Illustration

To test whether you understand the Chain of Responsibility pattern, cover the lefthand column of the table below and see if you can identify its players among the items from the illustrative example (Figure 8-1), as shown in the righthand column. Then check your answers against the lefthand column.

Client	The customer
Handlers	Clerks, supervisors, managersn
Successor	The next most senior handler
Request	The customer's request, query, or problem

As we'll see next, the most efficient implementation of the Chain of Responsibility pattern depersonalizes the Handlers, reducing them to the operations that they perform.

In terms of the design, there can be variations on Figure 8-2:

- The handlers may differ only in terms of the values of the state they store (e.g., a limit on an amount), or in the behavior of the Request method. In this case, the IHandler interface can be dispensed with, and the Handlers can still be instantiated in sufficient variety.
- The UML diagram implies only one instantiation of each Handler, resulting in one handler at each level in the chain. Practically speaking, there might be several identical handlers at certain levels (e.g., clerks in a bank). A different design is required to accommodate the latter case. This design is shown in the upcoming example code.

- What happens when a request reaches the end of the chain without finding a valid handler is an important design decision. The last object may handle the request in a default manner, or an exception may be thrown without the request being handled.
- However they are implemented, Handlers need to have a means for deciding whether they can rightly handle a request. There are several ways of implementing this decision, as we shall see.

Implementation

The simple theory code for the Chain of Responsibility pattern is shown in Example 8-1. The program creates five handlers that can deal with amounts up to 1,000 times their handler IDs. Each handler links to the one that can handle a higher amount. The handlers are created from highest to lowest in the loop at lines 28–31. The links point backward, and we get the structure shown in Figure 8-3.

Example 8-1. Chain of Responsibility pattern theory code

```

1  using System;
2
3  class ChainwithStatePattern {
4
5      // Chain of Responsibility Pattern    Judith Bishop   June 2007
6
7      class Handler {
8          Handler next;
9          int id;
10         public int Limit {get; set;}
11         public Handler (int id, Handler handler) {
12             this.id = id;
13             Limit = id*1000;
14             next = handler;
15         }
16         public string HandleRequest(int data) {
17             if (data < Limit)
18                 return "Request for " +data+" handled at level "+id;
19             else if (next!=null)
20                 return next.HandleRequest(data);
21             else return ("Request for " +data+" handled BY DEFAULT at level "+id);
22         }
23     }
24
25     static void Main () {
26
27         Handler start = null;
28         for (int i=5; i>0; i--) {
29             Console.WriteLine("Handler "+i+" deals up to a limit of "+i*1000);
30             start = new Handler(i, start);
31         }
32
33         int [] a = {50,2000,1500,10000,175,4500};
34         foreach (int i in a)

```

Example 8-1. Chain of Responsibility pattern theory code (continued)

```
35         Console.WriteLine(start.HandleRequest(i));
36     }
37
38 }
39 /* Output
40 Handler 5 deals up to a limit of 5000
41 Handler 4 deals up to a limit of 4000
42 Handler 3 deals up to a limit of 3000
43 Handler 2 deals up to a limit of 2000
44 Handler 1 deals up to a limit of 1000
45 Request for 50 handled at level 1
46 Request for 2000 handled at level 3
47 Request for 1500 handled at level 2
48 Request for 10000 handled BY DEFAULT at level 5
49 Request for 175 handled at level 1
50 Request for 4500 handled at level 5
51 */
```

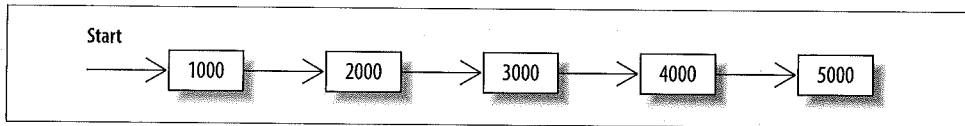


Figure 8-3. Chain of handlers

We'll start by exploring the implementation from the client's point of view. The client wants to call a handler with some data. For example:

```
handler(2000);
```

The crux of the Chain of Responsibility pattern is in lines 35 and then 20. Line 35 calls a handler, and line 20 can pass the request on to the next in line. The handler makes this decision based on conditions it can evaluate. Lines 18 and 21 show the other two outcomes: handling the request and handling it by default if the handler is the last in the chain.

The Chain of Responsibility pattern has an inherent limitation in that the request can get lost if the chain is not set up properly or if no handler is appropriate for the passed-in data. For this reason, the last handler has a default action. An alternative implementation, as mentioned earlier, would be to define a `ChainException` that the Handler can throw and the Client can catch. This approach is shown in the following example.

Example: Trusty Bank

To further explore the Chain of Responsibility pattern, we'll model the action of a bank that is careful about allowing withdrawals of large amounts. Trusty Bank's business rules for handling withdrawals are:

- Clerks can handle withdrawals of up to \$1,000.
- Supervisors can handle withdrawals of up to \$4,000.
- The bank manager can handle withdrawals of up to \$9,000.

Any amount larger than \$9,000 has to be divided into several withdrawals. The bank has several clerks on duty at any one time (up to 10), and usually 3 supervisors; there is only one manager.

In this example of the Chain of Responsibility pattern, we distinguish between levels in the chain and the number of object instances at each of those levels. The organizational structure we want is shown in Figure 8-4.

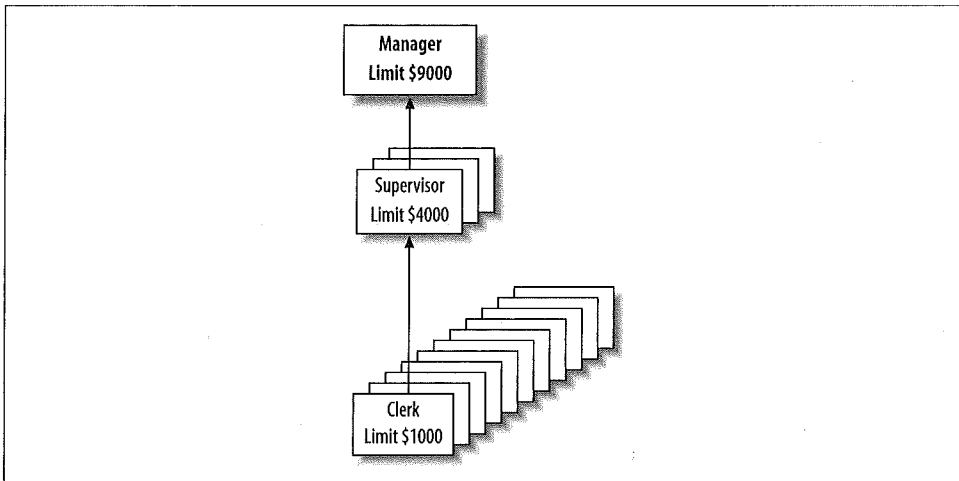


Figure 8-4. Trusty Bank organizational structure

We reflect this structure in two C# collections: one for the information about the limits and the number of positions, and one for the actual Handler objects used at runtime. The collections are:

```
static Dictionary <Levels, Structure> structure =
    new Dictionary <Levels, Structure> {
        {Levels.Manager, new Structure {Limit = 9000, Positions =1}},
        {Levels.Supervisor, new Structure {Limit = 4000, Positions =3}},
        {Levels.Clerk, new Structure {Limit = 1000, Positions =10}}};

static Dictionary <Levels, List<Handler>> handlersAtLevel =
    new Dictionary <Levels, List<Handler>> {
        {Levels.Manager, new List <Handler>()},
        {Levels.Supervisor, new List <Handler>()},
        {Levels.Clerk, new List <Handler>()}};
```

This excerpt uses two interesting C# features: *initializing* and *enumerated types*. We looked at initializing in Chapter 3, but I'll repeat the information here, more specifically for collections.

C# 3.0 Feature—Initializing Collections

In initializing a collection, we match the values with the required structure. If the items in any dimension of the structure are objects, `new` is required, and we open another set of brackets. If values are available, they can be inserted, as in:

```
{Levels.Clerk, new Structure {Limit = 500,  
                             Positions =10}},
```

Otherwise, an object constructor will do, as in:

```
{Levels.Clerk, new List <Handler>()},
```

If the structure is local to a method, its type can be inferred from the initialization and the `var` declaration is permitted, as in:

```
var structure = new Dictionary <Levels, Structure> {
```

cf. C# Language Specification Version 3.0, September 2007, Section 7.5.10.1-3

Both of the collections use an enumerated type: `Levels`. It is declared as:

```
enum Levels {Manager, Supervisor, Clerk}
```

Because enum constants can be associated with values, we could include the limits for the handler types with them, as in:

```
enum Levels (Manager = 9000, Supervisor = 4000, Clerk = 1000)
```

We'll revisit this idea in the upcoming "Exercises" section.

Having set up the structure, we then create a collection of three lists called `handlersAtLevel`. Each list contains the people who are at the level indicated, exactly as in Figure 8-4. The loop to create the handlers (requiring less writing) is:

```
foreach (Levels level in Enum.GetValues(typeof(Levels))) {  
    for (int i=0; i<structure[level].Positions; i++) {  
        handlersAtLevel[level].Add(new Handler(i, level));  
    }  
}
```

Thus, for each level, we pick up from the structure chart the number of positions required and instantiate that number of handlers, passing through an identifier and the level. Notice that this way of setting up the handlers differs from the theory code in Example 8-1 because there is no direct link to another handler; the link is deduced from the level plus a random number and found in the lists of `handlersAtLevel`.

The next step is to run the bank, accepting various requests, as in the theory example. Here's the loop:

C# Feature—Enumerated Types

An enumerated type (enum) specifies a set of named constants. The value of an enum is written out as if it were a string. For example:

```
enum Levels (Manager, Supervisor, Clerk)
Levels me = Manager;
Console.WriteLine((Levels) me);
```

Enum variables can be used in switch statements, and variables of the type can make use of comparison operations. In addition, the two operators ++ and -- go forward and backward in the set, and an error will occur if there is no value to move to. Enums can also be used in a foreach statement, as follows:

```
foreach (Levels level in Enum.GetValues(typeof(Levels))) {
```

This formulation is rather clumsy, but it's the best C# has available at present.

In addition, complicated methods are needed to detect whether a value is the first or last of a set.^a

Enum constants can have integer values associated with them. By default, these start at zero; however, casting back and forth to the integer type has to be done explicitly.

cf. *C# Language Specification Version 3.0*, September 2007, Section 14

^a See Marc Clifton's article "The Enumerable Enumerator" (November 2, 2006), available at <http://www.codeproject.com/csharp/EnumerableEnumerator.asp>.

```
int [] amounts = {50,2000,1500,10000,175,4500};
foreach (int amount in amounts) {
    try {
        int which = choice.Next(structure[Levels.Clerk].Positions);
        Console.WriteLine(handlersAtLevel
            [Levels.Clerk][which].HandleRequest(amount));
        AdjustChain();
    } catch (ChainException e) {
        Console.WriteLine("\nNo facility to handle a request of "+
            e.Data["Limit"]+
            "\nTry breaking it down into smaller requests\n");
    }
}
```

Here, we use a random number generator to pick a clerk and then call the chosen handler at the clerk level. Inside the chosen handler, the limit is checked. If the handler cannot deal with the amount, it uses the same technique to pick a supervisor, generating a number between 0 and 2. Given all this, the output from the program is:

```
1 Trusty Bank opens with
2 1 Manager(s) who deal up to a limit of 9000
3 3 Supervisor(s) who deal up to a limit of 4000
4 10 Clerk(s) who deal up to a limit of 1000
5
6 Approached Clerk 4. Request for 50 handled by Clerk 4
7 Approached Clerk 0. Request for 2000 handled by Supervisor 1
```

- 8 Approached Clerk 9. Request for 1500 handled by Supervisor 0
- 9 Approached Clerk 1.
- 10 No facility to handle a request of 10000
- 11 Try breaking it down into smaller requests
- 12
- 13 Approached Clerk 3. Request for 175 handled by Clerk 3
- 14 Approached Clerk 3. Request for 4500 handled by Manager 0
- 15 Approached Clerk 1. Request for 2000 handled by Supervisor 1

As this output shows, Clerks handled requests for \$50 and \$175 (lines 6 and 13), and three requests were sent up to Supervisors (not always the same one) on lines 7, 8, and 15. The Manager authorized one request (line 14) and rejected another (lines 9–11). When even the Manager cannot handle the request, the Handler interacts with the Client via an exception.

C# Feature—Exception Properties

C# exceptions are objects that can be created, thrown, and caught. Exceptions are caught in try/catch blocks; different catch clauses can catch different exceptions. Exceptions can carry information from the throwing object to the catching one. The following properties are useful in user programming:

Message

A string property

Data

An IDictionary that can hold key/value pairs

cf. C# Language Specification Version 3.0, September 2007, Section 16, and C# Class Library, the Exception Class

The full code for the Trusty Bank program is presented in Example 8-2.

Example 8-2. Chain of Responsibility pattern example code—Trusty Bank

```
using System;
using System.Collections.Generic;

class ChainPatternExample {
    // Chain of Responsibility Pattern Example          Judith Bishop  Sept 2007
    // Sets up the Handlers as level-based structure
    // Makes use of a user-defined exception if the end of the chain is reached

    class Handler {
        Levels level;
        int id;
        public Handler (int id, Levels level) {
            this.id = id;
            this.level = level;
        }
    }
}
```

Example 8-2. Chain of Responsibility pattern example code—Trusty Bank (continued)

```
public string HandleRequest(int data) {
    if (data < structure[level].Limit) {
        return "Request for " +data+" handled by "+level+ " "+id;
    }
    else if (level > First) {
        Levels nextLevel = --level;
        int which = choice.Next(structure[nextLevel].Positions);
        return handlersAtLevel[nextLevel][which].HandleRequest(data);
    }
    else {
        Exception chainException = new ChainException();
        chainException.Data.Add("Limit", data);
        throw chainException;
    }
}

public class ChainException : Exception {
    public ChainException() {}
}

void AdjustChain() {}

enum Levels {Manager, Supervisor, Clerk};
static Random choice = new Random(11);
static Levels First {
    get { return ((Levels[])Enum.GetValues(typeof(Levels)))[0]; }
}

static Dictionary <Levels,Structure> structure =
    new Dictionary <Levels, Structure> {
        {Levels.Manager, new Structure {Limit = 9000, Positions =1}},
        {Levels.Supervisor, new Structure {Limit = 4000, Positions =3}},
        {Levels.Clerk, new Structure {Limit = 1000, Positions =10}};
}

static Dictionary <Levels, List<Handler>> handlersAtLevel =
    new Dictionary <Levels, List<Handler>> {
        {Levels.Manager, new List <Handler>()},
        {Levels.Supervisor, new List <Handler>()},
        {Levels.Clerk, new List <Handler>()};
}

class Structure {
    public int Limit {get; set;}
    public int Positions {get; set;}
}

void RunTheOrganization () {

    Console.WriteLine("Trusty Bank opens with");
    foreach (Levels level in Enum.GetValues(typeof(Levels))) {
        for (int i=0; i<structure[level].Positions; i++) {
            handlersAtLevel[level].Add(new Handler(i, level));
        }
    }
}
```

Example 8-2. Chain of Responsibility pattern example code—Trusty Bank (continued)

```
    }
    Console.WriteLine(structure[level].Positions+ " "+ level+
        "(s) who deal up to a limit of " + structure[level].Limit);
}
Console.WriteLine();

int [] amounts = {50,2000,1500,10000,175,4500,2000};
foreach (int amount in amounts) {
    try {
        int which = choice.Next(structure[Levels.Clerk].Positions);
        Console.Write("Approached Clerk "+which+". ");
        Console.WriteLine(handlersAtLevel[Levels.Clerk][which].
            HandleRequest(amount));
        AdjustChain();
    } catch (ChainException e) {
        Console.WriteLine("\nNo facility to handle a request of "+
            e.Data["Limit"]+
            "\nTry breaking it down into smaller requests\n");
    }
}

static void Main () {
    new ChainPatternExample().RunTheOrganization();
}
}
```

In the preceding example, the handlers were all instantiated from the same class. But according to the UML diagram (Figure 8-2), the classes can be different, as long as they conform to an interface with a well-defined Request method. In this case, using delegates to set up the chain of request methods might be a useful technique.

Use

The Chain of Responsibility pattern is used in tools that handle user events. It is often employed in conjunction with events and exceptions.

Use the Chain of Responsibility pattern when...

You have:

- More than one handler for a request
- Reasons why a handler should pass a request on to another one in the chain
- A set of handlers that varies dynamically

You want to:

- Retain flexibility in assigning requests to handlers

Exercises

1. Change the enumerated type in the Trusty Bank example so that each constant has an associated value that refers to its limit. Rework the program accordingly.
2. In the Trusty Bank example, set up a completely different Handler class called HeadOffice. Extend the system to enable requests to go from the Manager to HeadOffice for approval.
3. Exception handlers work in chains and are themselves an example of the Chain of Responsibility pattern. As an exercise, implement the notion of an exception-handling mechanism without using C#'s built-in exception-handling mechanism.

Command Pattern

Role

The Command pattern creates distance between the client that requests an operation and the object that can perform it. This pattern is particularly versatile. It can support:

- Sending requests to different receivers
- Queuing, logging, and rejecting requests
- Composing higher-level transactions from primitive operations
- Redo and Undo functionality

Illustration

The Command pattern is used in the menu systems of many well-known applications. An example of such an interface is shown in Figure 8-5 (which happens to be the program editor I usually use). Note that there are different insert and copy operations.

The figure shows two ways in which commands can be activated: select them from a drop-down textual menu, or click on the icons that represent each of the operations. In the top menu, there could be a receiver that handles word processing cut-and-paste-type operations, and another for brace matching and changing case. For most of these commands, Undo and Redo functions will also be appropriate; however, some operations are indicated in gray, which means they cannot be undone or repeated. Furthermore, in systems such as this one, there is usually a way of stringing together some commands so that they can be executed in a batch later. These so-called *macro* commands are common in graphics and photo-editing programs.