21

CHAIN OF RESPONSIBILITY

This pattern was previously described in GoF95.

DESCRIPTION

The Chain of Responsibility pattern (CoR) recommends a low degree of coupling between an object that sends out a request and the set of potential request handler objects.

When there is more than one object that can handle or fulfill a client request, the CoR pattern recommends giving each of these objects a chance to process the request in some sequential order. Applying the CoR pattern in such a case, each of these potential handler objects can be arranged in the form of a chain, with each object having a pointer to the next object in the chain. The first object in the chain receives the request and decides either to handle the request or to pass it on to the next object in the chain. The request flows through all objects in the chain one after the other until the request is handled by one of the handlers in the chain or the request reaches the end of the chain without getting processed.

As an example, if A $\rightarrow$ B $\rightarrow$ C are objects capable of handling the request, in this order, then A should handle the request or pass on to B without determining whether B can fulfill the request. Upon receiving the request, B should either handle it or pass on to C. When C receives the request, it should either handle the request or the request falls off the chain without getting processed. In other words, a request submitted to the chain of handlers may not be fulfilled even after reaching the end of the chain.

The following are some of the important characteristics of the CoR pattern:

- The set of potential request handler objects and the order in which these objects form the chain can be decided dynamically at runtime by the client depending on the current state of the application.
- A client can have different sets of handler objects for different types of requests depending on its current state. Also, a given handler object may need to pass on an incoming request to different other handler objects depending on the request type and the state of the client application. For these communications to be simple, all potential handler objects should provide a consistent interface. In Java this can be accomplished by having

different handlers implement a common interface or be subclasses of a common abstract parent class.

- The client object that initiates the request or any of the potential handler objects that forward the request do not have to know about the capabilities of the object receiving the request. This means that neither the client object nor any of the handler objects in the chain need to know which object will actually fulfill the request.
- Request handling is not guaranteed. This means that the request may reach the end of the chain without being fulfilled. The following example presents a scenario where a purchase request submitted to a chain of handlers is not approved even after reaching the end of the chain.

EXAMPLE

Let us consider an application to simulate the purchase request (PR) authorization process in a typical organization. In general, a PR needs to be authorized by an appropriate management representative before an order to a vendor can be created. Let us consider an organization with four levels of management personnel listed in Table 21.1 who can authorize a PR with an amount less than their authorization limit.

We can define different classes (Listing 21.1) to represent each management level listed in Table 21.1.

Let us define a `PurchaseRequest` class (Figure 21.1 and Listing 21.2) that represents a purchase request.

A given PR could be authorized or handled by any of the management representatives listed in Table 21.1. In other words, each of the four classes representing different levels of management is a potential handler for a given PR and hence it is not advisable to tie a `PurchaseRequest` instance to any of the handlers. By using the CoR pattern, a low-coupling association between a `PurchaseRequest` object and the set of potential handler objects can be achieved.

Applying the CoR pattern, let us define an abstract `PRHandler` class (Listing 21.3) that declares the common interface to be offered by all of the potential PR handlers (Figure 21.2).

Each of the handlers can now be redesigned as a subclass of the abstract `PRHandler` class (Listing 21.4). As part of its implementation, each handler object compares the PR amount with the authorization limit of the management representative it represents. If the PR amount is less than the authorization limit, it

Table 21.1 Levels of PR Authorization

Management Level	*Authorization Limit*
Branch Manager	$25,000
Regional Director	$100,000
Vice President	$200,000
President and COO	$400,000

Listing 21.1 Classes Representing Different Management Levels

```
class BranchManager {
  static double LIMIT = 25000;
            …
            …
}//End of class
class RegionalDirector {
  static double LIMIT = 100000;
            …
            …
}//End of class
class VicePresident {
  static double LIMIT = 200000;
            …
            …
}//End of class
class PresidentCOO {
  static double LIMIT = 400000;
            …
            …
}//End of class
```

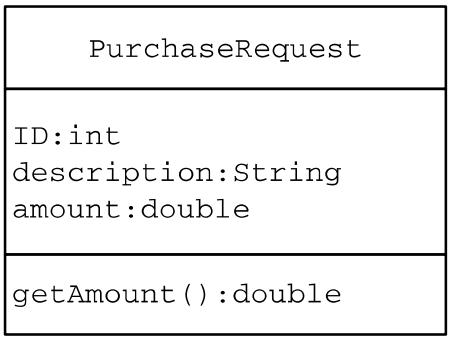

Figure 21.1 `PurchaseRequest` Class Representation

authorizes the PR. If not, it passes the PR authorization request to the next handler in the chain.

To authorize a PR, a client (Listing 21.5) would:

1. Create a set of potential PR authorization request handler objects and arrange them in an ascending order by authorization limit. Connect each handler to the next handler using the `setNextHandler(PRHandler)` method. This results in a chain of potential PR authorization request handlers (Figure 21.3).

Listing 21.2 `PurchaseRequest` Class

```
class PurchaseRequest {
  private int ID;
  private String description;
  private double amount;
  public PurchaseRequest(int id, String desc, double amt) {
    ID = id;
    description = desc;
    amount = amt;
  }
  public double getAmount() {
    return amount;
  }
  public String toString() {
    return ID + ":" + description;
  }
}
```

Listing 21.3 Abstract `PRHandler` Class

```
public abstract class PRHandler {
  private PRHandler nextHandler;
  private String handlerName;
  public PRHandler(String name) {
    handlerName = name;
  }
  public String getName() {
    return handlerName;
  }
  public abstract boolean authorize(PurchaseRequest request);
  public PRHandler getNextHandler() {
    return nextHandler;
  }
  public void setNextHandler(PRHandler handler) {
    nextHandler = handler;
  };
}
```

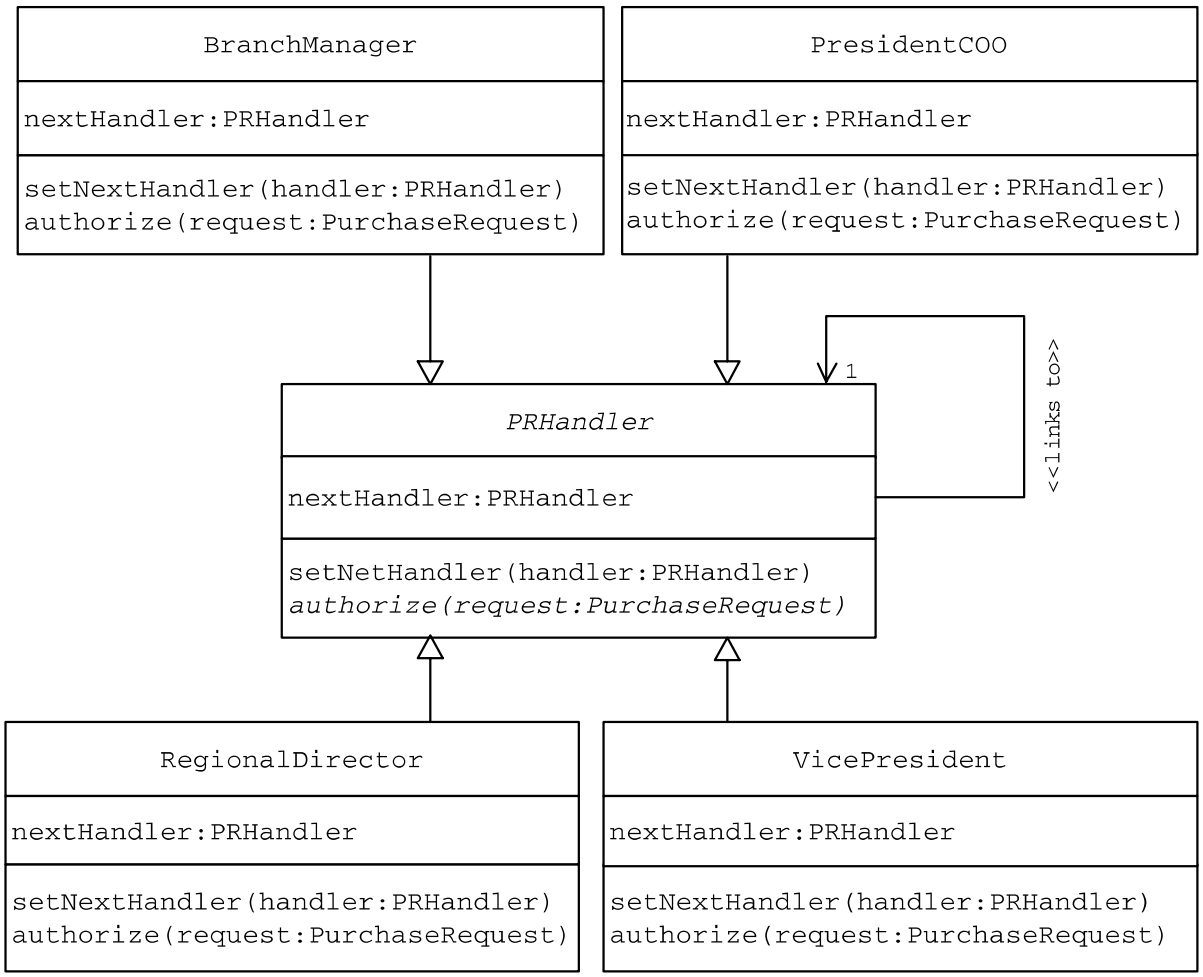

Figure 21.2 Purchase Request Approver Hierarchy

2. Send a PR authorization request to the first `PRHandler` object in the chain by invoking the `authorize` method on that object, passing the purchase request as an argument. As can be seen from the implementation of different `PRHandler` subclasses, a PR is authorized if the PR amount is less than the authorization limit of a specific handler. Otherwise, the authorization request is passed on to the next potential handler in the chain. If the PR is authorized by one of the handlers, it is not passed on to the next handler in the chain. The PR authorization is not guaranteed in this example. If the request reaches the last handler and the PR amount is higher than the authorization limit of the last handler, an appropriate message is displayed and the PR remains unauthorized.

When the client `PRManager` is run, output similar to the following is displayed:

```
Branch Manager Robin has authorized the PR - 1:Office Supplies
V.P. Kate has authorized the PR - 2:HardWare Procurement
PR - 3:AD Campaign couldn't be authorized.
Executive Board needs to be consulted for approval
reason: Amount too large
```

The sequence diagram in Figure 21.4 shows the message flow when a $150,000 PR authorization request is sent to the chain of potential handler objects.

Listing 21.4 `PRHandler` Concrete Subclasses

```
class BranchManager extends PRHandler {
  static double LIMIT = 25000;
  public BranchManager(String name) {
    super(name);
  }
  public boolean authorize(PurchaseRequest request) {
    double amount = request.getAmount();
    if (amount <= LIMIT) {
      System.out.println(" Branch Manager " + getName() +
                         " has authorized the PR - " + request);
      return true;
    } else {
      //forward the request to the next handler
      return getNextHandler().authorize(request);
    }
  }
}//End of class
class RegionalDirector extends PRHandler {
  static double LIMIT = 100000;
  public RegionalDirector(String name) {
    super(name);
  }
  public boolean authorize(PurchaseRequest request) {
    double amount = request.getAmount();
    if (amount <= LIMIT) {
      System.out.println(" Regional Director " + getName() +
                         " has authorized the PR - " +
                         request);
      return true;
    } else {
      //forward the request to the next handler
      return getNextHandler().authorize(request);
    }
  }
}//End of class
```

(continued)

Listing 21.4 `PRHandler` Concrete Subclasses (Continued)

```
class VicePresident extends PRHandler {
  static double LIMIT = 200000;
  public VicePresident(String name) {
    super(name);
  }
  public boolean authorize(PurchaseRequest request) {
    double amount = request.getAmount();
    if (amount <= LIMIT) {
      System.out.println(" V.P. " + getName() +
                         " has authorized the PR - " + request);
      return true;
    } else {
      //forward the request to the next handler
      return getNextHandler().authorize(request);
    }
  }
}//End of class
class PresidentCOO extends PRHandler {
  static double LIMIT = 400000;
  public PresidentCOO(String name) {
    super(name);
  }
  public boolean authorize(PurchaseRequest request) {
    double amount = request.getAmount();
    if (amount <= LIMIT) {
      System.out.println(" President & COO " + getName() +
                         " has authorized the PR - " + request);
      return true;
    } else {
      System.out.println("PR - " + request +
                         " couldn't be authorized.\n " +
                         "Executive Board needs to be " +
                         "consulted for approval \n" +
                         "reason: Amount too large");
      return false;
    }
  }
}//End of class
```

Listing 21.5 Client `PRManager` Class

```
public class PRManager {
  private BranchManager branchManager;
  private RegionalDirector regionalDirector;
  private VicePresident vicePresident;
  private PresidentCOO coo;
  public static void main(String[] args) {
    PRManager manager = new PRManager();
    manager.createAuthorizationFlow();
    PurchaseRequest request =
      new PurchaseRequest(1, "Office Supplies”,10000);
    manager.branchManager.authorize(request);
    request = new PurchaseRequest(2, "HardWare Procurement”,
              175000);
    manager.branchManager.authorize(request);
    request = new PurchaseRequest(3, "AD Campaign”,800000);
    manager.branchManager.authorize(request);
  }
  public void createAuthorizationFlow() {
    branchManager = new BranchManager("Robin");
    regionalDirector = new RegionalDirector("Oscar");
    vicePresident = new VicePresident("Kate");
    coo = new PresidentCOO("Drew");
    branchManager.setNextHandler(regionalDirector);
    regionalDirector.setNextHandler(vicePresident);
    vicePresident.setNextHandler(coo);
  }
}//End of class
```

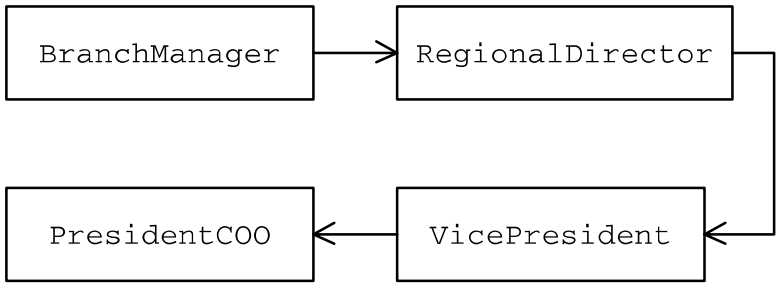

Figure 21.3 Chain of PR Authorization Request Handlers

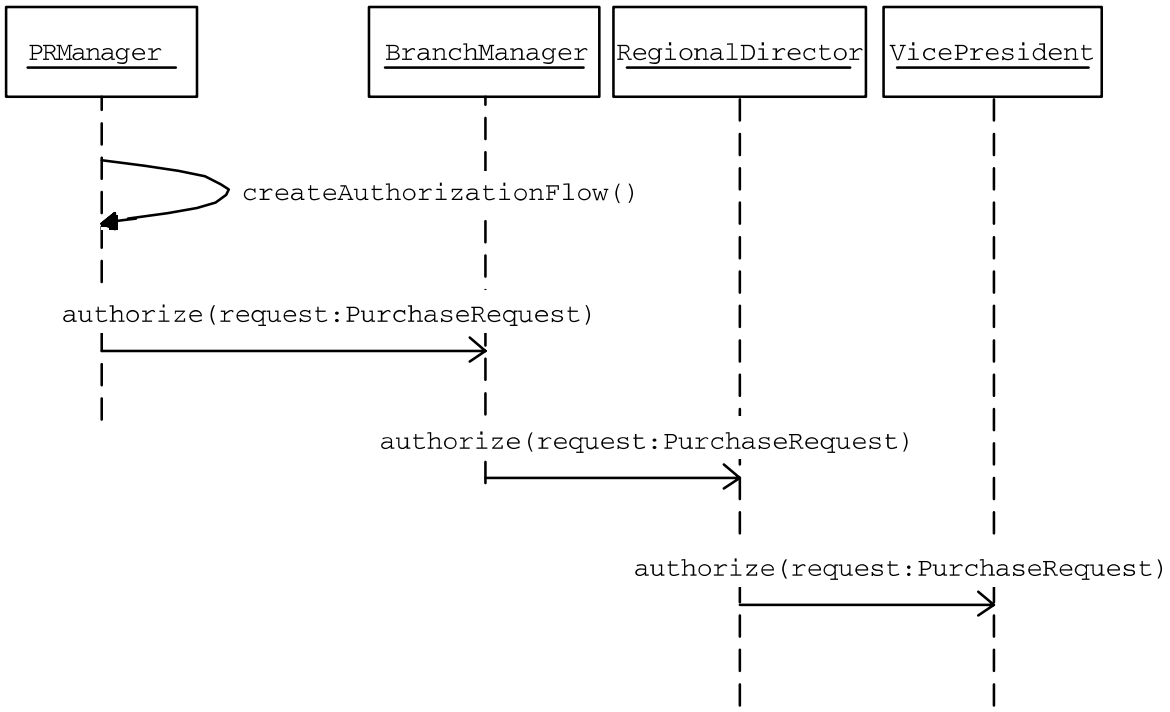

Figure 21.4 Purchase Request Authorization: Message Flow

PRACTICE QUESTIONS

1. In the example above, a given PR always needs to be approved by only one person in the chain with a higher approval limit than the PR amount. For example, a PR for $50,000 needs approval from a regional director (with approval limit $100,000). It does not need the approval of a branch manager (with approval limit $25,000). In general, if a PR is to be approved by one person in the approval chain, it does not need the approval of any other person in the chain. In some cases, it may be required that a given purchase request be approved by all individuals with the approval limit less than the purchase request amount until it is approved by an individual with a higher approval limit than the PR amount. Modify the example application to implement this purchase request approval process.
2. Let us consider an ISP (Internet service provider) with three levels of technical support as follows:
   a. *Service Level I (Basic)* — Aimed at resolving basic connectivity problems such as incorrect/forgotten passwords, incorrect dial-up number, etc.
   b. *Service Level II* — When the Basic Service Level I support team cannot resolve a problem, it will be sent to the Service Level II team. For problem resolution, the Service Level II team assumes the user to have a good understanding of computer concepts.
   c. *Service Level III* — When the Service Level I and II teams cannot resolve a problem, it will be sent to the Service Level III team. A technician schedules an appointment with the user for problem resolution at the user site.

Create an application using the CoR pattern to simulate the three-layer technical support structure explained above.

3. Let us consider an IT consulting firm with three levels of resource coordinators as follows:
   - Local resource manager
   - Regional resource coordinator
   - Corporate resource director

   Whenever a consultant becomes available, that consultant's profile is first sent to the local resource manager to see if there is any requirement locally that matches the skill set of the consultant. If there is no requirement that matches with the consultant's skill set, the consultant's details are sent to the regional resource coordinator.

   The regional resource coordinator, with access to a much broader area of current and prospective requirements within the region, will then look for a possible match for the consultant's skill set. If there is no match, the consultant's data is sent to the corporate resource director.

   The corporate resource director will be able to look for a matching assignment for the consultant's skill set across all regions of the company operation. Create an application using the CoR pattern to simulate this process.