Understanding and Implementing Chain of Responsibility Pattern in C#

Rahul Rajat Singh, 16 Nov 2012

4.36 (9 votes) Rate:

This article talks about the Chain of responsibility pattern.

Download demo - 12.1 KB

Introduction

This article talks about the Chain of responsibility pattern. We will try to see when this pattern can be useful and what are the benefits of using this pattern. We will also look at a rudimentary implementation of Chain of responsibility pattern in C#.

Background

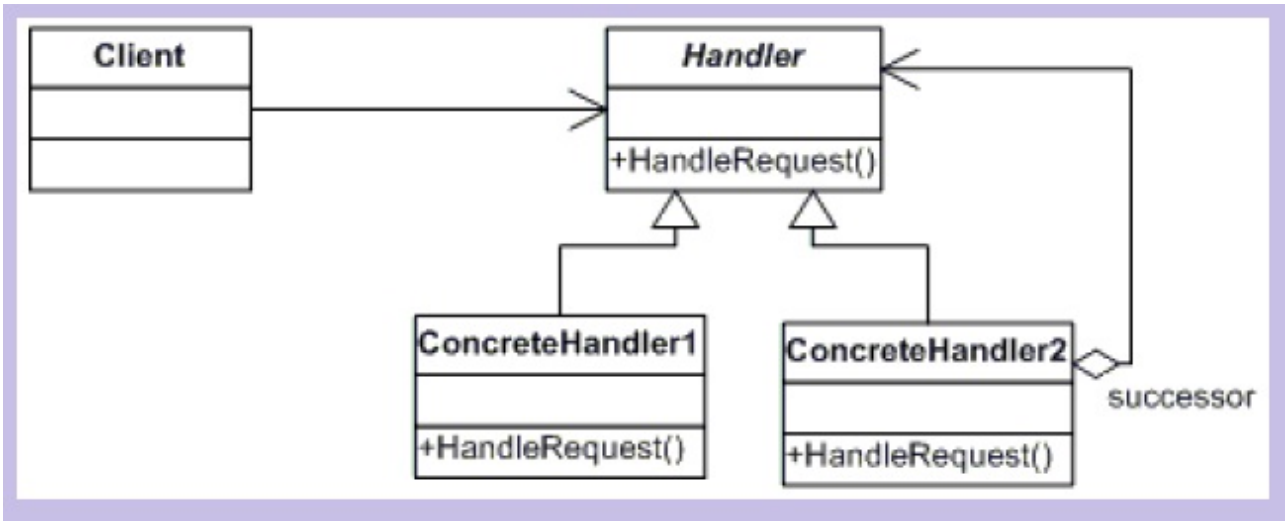
We have seen the scenarios where one class need to know about the status change of another class while looking at the Observer pattern. In Observer pattern a class can register itself to another class to get the notification whenever the state of other class is changing. The notification will come in form of an event and the listener class could then decide what action it needs to take.

Now if we take this scenario a step further where the class listening to events will take the decision based on some conditions. If the condition is not met then it will further pass on this event to another object that could handle this event.

So in this scenario we have a series of objects that can handle the event based on some criteria. These all object will pass on the event to others in a sequential manner. Whether this class could take the action or it need to send the event further is the logic that will be contained in this class. All the classes that can handle the event are chained together i.e. ever class contains a handle to its successor to which it can pass on the event to.

Chain of responsibility pattern is meant for such scenarios. In this pattern an object will listen for an event, when this event occurs it handles the event. if this object is capable of taking some action, it will otherwise it will propagate the event to another object in line which could in turn handle the event or propagate it further. This pattern is very useful in implementing workflow kind of scenarios.

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



To understand this class diagram lets look at each class.

- `Handler`: This is the interface or abstract class that all the classes who could handle the event should implement.
- `ConcreteHandler`: There are the classes that are capable of either handling the event or propogating it further.
- `Client`: This is the class that defines the chain i.e. successors for all `ConcreteHandlers` and this is the class that initiates the request to a `ConcreteHandler`.

Using the code

To understand this pattern better, lets try to implement a small rudimentary work flow application.

Lets say we have an organization where Team members when apply for leave, the request goes to the Team Leader. Team leader can approve all the leave request of less than 10 days. If the leave request is of more than 10 days then the request will be passed on to the Project Leader. Project leader is able to approve leaves of upto 20 days. If the leave is applied for more than 20 days then this requests will be passed to the HR. HR can approve upto 30 days of leave. If the leave is of more than 30 days then the leave application cannot be approved by the system and it needs a manual process for approval.

Now with the above requirement, if we try to implement the Chain of responsibility pattern then we need `ConcreteHandlers` for Team Leader, Project Leader and HR. These `ConcreteHandlers` will be chained together so that the request can pass from `TeamLeader` to `ProjectLeader` to `HR`.

Also, we need an abstract class that can contain the common functionality like keeping track oc successor object, initiating the request and the eventing mechanism. Lets call this class `Employee`. The `ConcreteHandler` will contain logic specific to the concrete handlers.

Lets start by looking at the `Handler` abstract class i.e. `Employee` class.

Hide Shrink ▲ Copy Code

```
public abstract class Employee
{
    // Every employee will have a supervisor
    protected Employee supervisor;

    // Event mechanism to know whenever a leave has been applied
    public delegate void OnLeaveApplied(Employee e, Leave l);
    public event OnLeaveApplied onLeaveApplied = null;

    // This will invoke events when the leave will be applied
    // i.e. the actual item will be handed over to the hierarchy of
    // concrete handlers.
    public void LeaveApplied(Employee s, Leave leave)
    {
        if (onLeaveApplied != null)
        {
            onLeaveApplied(this, leave);
        }
    }

    // This is the function which concrete handlers will use to take
```

```
    // action, if they are able to take actions.
    public abstract void ApproveLeave(Leave leave);

    // getter to get the supervisor of current employee
    public Employee Supervisor
    {
        get
        {
            return supervisor;
        }
        set
        {
            supervisor = value;
        }
    }

    // Using this we can apply for leave
    public void ApplyLeave(Leave l)
    {
        LeaveApplied(this, l);
    }
}
```

This `Handler` class is responsible for:

1. Keeping track of successors for this object
2. Implementing the eventing mechanism to notify and propagate the request event.
3. Initiate the request.

Now since our Handler class is ready, lets look at the `ConcreteHandlers` one by one. Lets start with `TeamLeader` class.

Hide Shrink Copy Code

```
public class TeamLeader : Employee
{
    // team leas can only approve upto 7 days of leave
    const int MAX_LEAVES_CAN_APPROVE = 10;

    // in constructor we will attach the event handler that
    // will check if this employee can process or he need to
    // pass on to next employee
    public TeamLeader()
    {
        this.onLeaveApplied += new OnLeaveApplied(TeamLeader_onLeaveApplied);
    }

    // in this function we will check if this employee can
    // process or he need to pass on to next employee
    void TeamLeader_onLeaveApplied(Employee e, Leave l)
    {
        // check if we can process this request
        if (l.NumberOfDays < MAX_LEAVES_CAN_APPROVE)
        {
            // process it on our level only
            ApproveLeave(l);
        }
        else
        {
            // if we cant process pass on to the supervisor
            // so that he can process
            if (Supervisor != null)
            {
                Supervisor.LeaveApplied(this, l);
            }
        }
    }

    // If we can process lets show the output
    public override void ApproveLeave(Leave leave)
    {
        Console.WriteLine("LeaveID: {0} Days: {1} Approver: {2}",
            leave.LeaveID, leave.NumberOfDays, "Team Leader");
    }
}
```

What this class is doing is:

1. Checking of this class could take the action i.e. leave applied is less than 10 days.
2. If this class could take the action then show the response to the user.
3. If this class is not able to take the action then pass on the request to the `ProjectLeader` class i.e. its successor.

Now lets look at the `ProjectLeader` class.

Hide Shrink Copy Code

```
class ProjectLeader : Employee
{
    const int MAX_LEAVES_CAN_APPROVE = 20;

    // in constructor we will attach the event handler that
    // will check if this employee can process or he need to
    // pass on to next employee
    public ProjectLeader()
    {
        this.onLeaveApplied += new OnLeaveApplied(ProjectLeader_onLeaveApplied);
    }

    // in this function we will check if this employee can
    // process or he need to pass on to next employee
    void ProjectLeader_onLeaveApplied(Employee e, Leave l)
    {
        // check if we can process this request
        if (l.NumberOfDays < MAX_LEAVES_CAN_APPROVE)
        {
            // process it on our level only
            ApproveLeave(l);
        }
        else
        {
            // if we cant process pass on to the supervisor
            // so that he can process
            if (Supervisor != null)
            {
                Supervisor.LeaveApplied(this, l);
            }
        }
    }

    // If we can process lets show the output
    public override void ApproveLeave(Leave leave)
    {
        Console.WriteLine("LeaveID: {0} Days: {1} Approver: {2}",
            leave.LeaveID, leave.NumberOfDays, "Project Leader");
    }
}
```

What this class is doing is:

1. Checking of this class could take the action i.e. leave applied is less than 20 days.
2. If this class could take the action then show the response to the user.
3. If this class is not able to take the action then pass on the request to the `HR` class i.e. its successor.

Now lets look at the `HR` class.

Hide Shrink Copy Code

```
class HR : Employee
{
    const int MAX_LEAVES_CAN_APPROVE = 30;

    // in constructor we will attach the event handler that
    // will check if this employee can process or
    // some other action is needed
    public HR()
    {
        this.onLeaveApplied += new OnLeaveApplied(HR_onLeaveApplied);
    }
```

```
    // in this function we will check if this employee can
    // process or some other action is needed
    void HR_onLeaveApplied(Employee e, Leave l)
    {
        // check if we can process this request
        if (l.NumberOfDays < MAX_LEAVES_CAN_APPROVE)
        {
            // process it on our level only
            ApproveLeave(l);
        }
        else
        {
            // if we cant process pass on to the supervisor
            // so that he can process
            if (Supervisor != null)
            {
                Supervisor.LeaveApplied(this, l);
            }
            else
            {
                // There is no one up in hierarchy so lets
                // tell the user what he needs to do now
                Console.WriteLine("Leave application suspended, Please contact HR");
            }
        }
    }

    // If we can process lets show the output
    public override void ApproveLeave(Leave leave)
    {
        Console.WriteLine("LeaveID: {0} Days: {1} Approver: {2}",
            leave.LeaveID, leave.NumberOfDays, "HR");
    }
}
```

What this class is doing is:

1. Checking of this class could take the action i.e. leave applied is less than 30 days.
2. If this class could take the action then show the response to the user.
3. If this class is not able to take the action then let the user know that he needs to have a manual discussion and his request has been suspended.

The actual action item i.e. the leave is encapsulated into a class for better modularity, so before running the application lets see how this `Leave` class looks like

Hide Copy Code

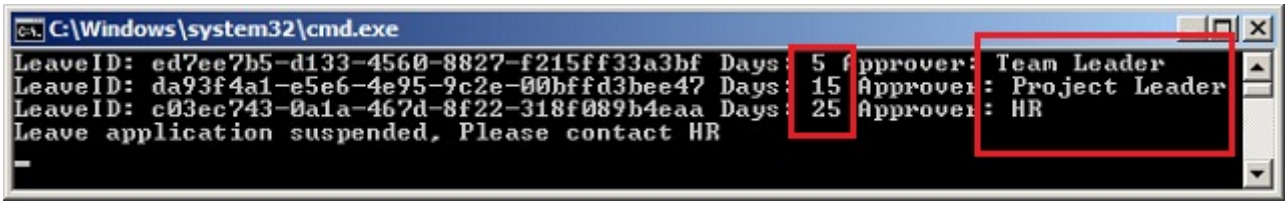
```
// This is the actual Item that will be used by the concretehandlers
// to determine whther they can act upon this request or not
public class Leave
{
    public Leave(Guid guid, int days)
    {
        leaveID = guid;
        numberOfDays = days;
    }

    Guid leaveID;

    public Guid LeaveID
    {
        get { return leaveID; }
        set { leaveID = value; }
    }
    int numberOfDays;

    public int NumberOfDays
    {
        get { return numberOfDays; }
        set { numberOfDays = value; }
    }
}
```

And finally we need the `Client` that will set the successor chain and initiate the request.

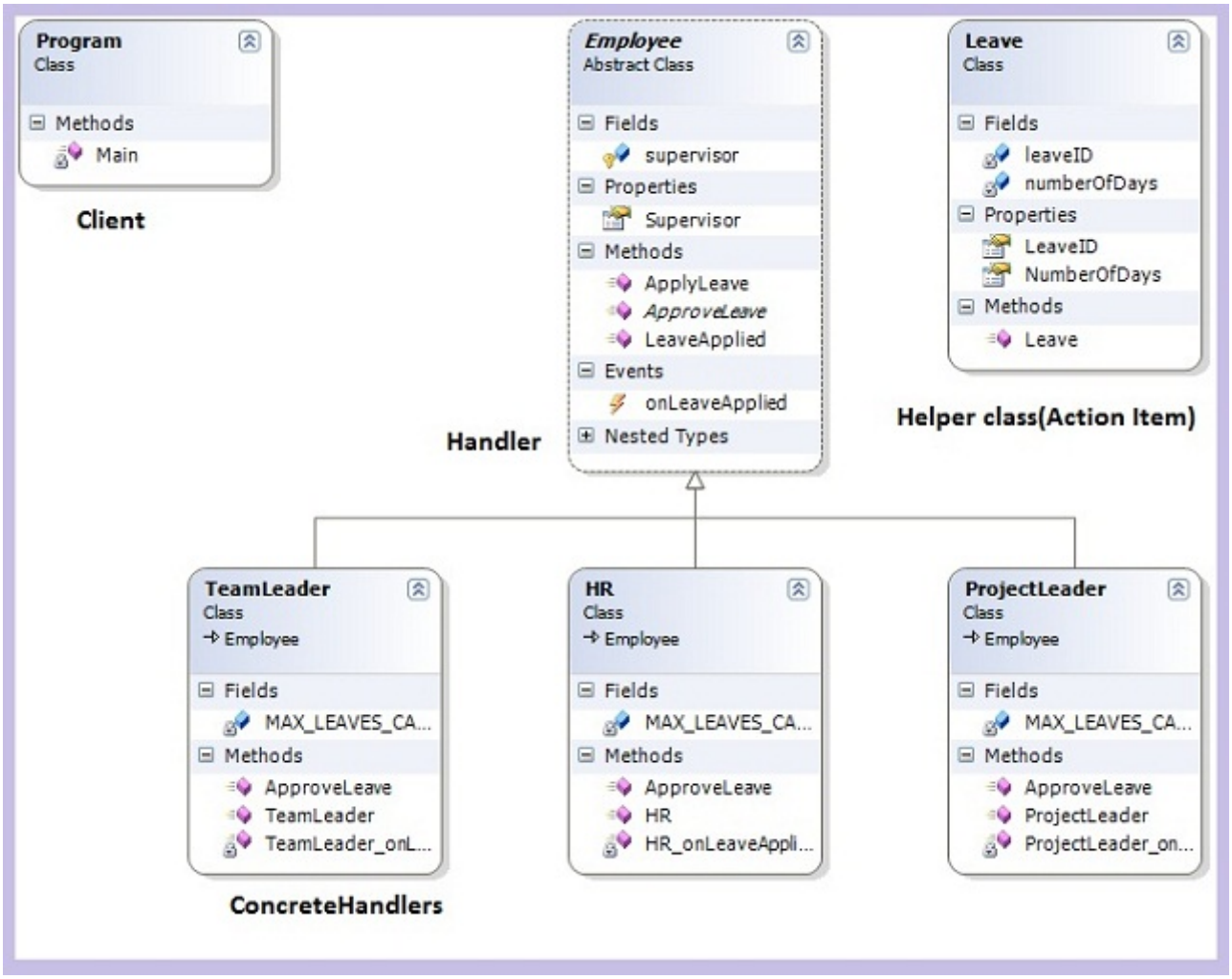
Hide Copy Code

```
class Program
{
    static void Main(string[] args)
    {
        // lets create employees
        TeamLeader tl = new TeamLeader();
        ProjectLeader pl = new ProjectLeader();
        HR hr = new HR();

        // Now lets set the hierarchy of employees
        tl.Supervisor = pl;
        pl.Supervisor = hr;

        // Now lets apply 5 day leave my TL
        tl.ApplyLeave(new Leave(Guid.NewGuid(), 5));

        // Now lets apply 15 day leave my TL
        tl.ApplyLeave(new Leave(Guid.NewGuid(), 15));

        // Now lets apply 25 day leave my TL
        tl.ApplyLeave(new Leave(Guid.NewGuid(), 25));

        // Now lets apply 35 day leave my TL
        tl.ApplyLeave(new Leave(Guid.NewGuid(), 35));

        Console.ReadLine();
    }
}
```

So we can see that this Main function is creating the chain by setting the successors for each class and is initiating the request to `TeamLeader` class. One request for each scenario has been made. When we run this application.



So we can see that each request get passed on and processed by the respective class based on the number of days applied for. Before wrapping up let us look at the class diagram for our application and compare it with the original GoF diagram.



Point of interest

In this article, we tried to look at the Chain of responsibility pattern. We saw when this pattern could be useful, what are the benefits of using this pattern and how can we have a sample rudimentary implementation for this pattern in C#. This article has been written from a beginner's perspective. I hope this has been informative.

History

- **16 November 2012**: First version

License

Share

About the Author

Rahul Rajat Singh
Software Developer (Senior)
India

Follow this Member

I Started my Programming career with C++. Later got a chance to develop Windows Form applications using C#. Currently using C#, ASP.NET & ASP.NET MVC to create Information Systems, e-commerce/e-governance Portals and Data driven websites.

My interests involves Programming, Website development and Learning/Teaching subjects related to Computer Science/Information Systems. IMO, C# is the best programming language and I love working with C# and other Microsoft Technologies.

- Microsoft Certified Technology Specialist (MCTS): Web Applications Development with Microsoft .NET Framework 4
- Microsoft Certified Technology Specialist (MCTS): Accessing Data with Microsoft .NET Framework 4
- Microsoft Certified Technology Specialist (MCTS): Windows Communication Foundation Development with Microsoft .NET Framework 4

If you like my articles, please visit my website for more: www.rahulrajatsingh.com[^]

- Microsoft MVP 2015

Comments and Discussions

Add a Comment or Question

Search Comments

First Prev Next

Like

Gena H 26-Mar-14 11:28

Couple of things I'd change...
KeithAMS 27-Nov-12 18:13

Re: Couple of things I'd change...
Rahul Rajat Singh 29-Nov-12 6:37

Re: Couple of things I'd change...
sandeshjkota 19-Nov-14 15:43

My vote of 5
SergoT 20-Nov-12 19:03

Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.