# SSE 661

# Software Architecture

# Project #3

by

Jason Payne

June 17, 2015

## TABLE OF CONTENTS

| Topics Covered | Topic Examples |
|---|---|
| Software System Design Patterns | • Chain of Responsibility<br>• Forwarder-Receiver |

## 1. Chain of Responsibility Pattern

The Chain of Responsibility (CoR) design pattern is useful for situations that necessitate that a request (or event) be handled by multiple handlers. It is considered a behavioral pattern because it defines the behavior for controlling communications between classes or entities. As the name indicates, a series of handlers are created as a linked list, or chain, which promotes loose coupling between the client application and the handlers.

Some examples of systems utilizing the CoR pattern include:

- C# exception handling system – when an exception is thrown, the method that caused the exception is given the chance to process it, via a try-catch block. If no suitable catch is available, the exception moves up to the calling method, which may include a try-catch. This continues until the exception is handled or until there are no more possible handlers.
- Vending machines – the CoR pattern is used in the software that controls the coin slot. The coin is weighed and measured and these dimensions are passed into the first link of the chain of handlers. If a handler can process the coin, the value is added to the current credit for the machine and the coin is retained. If the coin cannot be handled it is rejected.
- Windows UI events – a CoR is used to handle mouse and keyboard events.

### 1.1 Step-by-Step Design

The application that will be used to exercise the CoR pattern is based upon a video game framework. This framework will need a way to efficiently handle the gameplay capabilities of the player as they progress through the game gaining various upgrades (or downgrades) to weapons. The first natural instinct in the design is to create a single event handler for the case where a player receives a skills upgrade/downgrade. While that would most likely work for this specific game (and basic scenario), it may not work for other games that would want to handle the event in different ways. It would also not work at all if the handlers were to be determined dynamically (for example, a secret is unlocked in the game that allows the player to receive two upgrades at a time instead of one). Again, a single event handler could handle such an event, but what if other changes are requested of the handling mechanism? Code in the handler – and possibly the client – would have to be altered as the requirements changed. With the CoR pattern, the handlers circumvent this issue by allowing a way for their chained handlers to be rearranged at run-time (similar to the way that objects can change behaviors in the Decorator pattern).

Figure 1 illustrates the overall design of the CoR pattern used for this application and Appendix A provides the implementation of this design. The request is passed to the first handler in the chain (determined at run-time), which will either process the request or pass it on to its successor (the next handler, or link, in the chain). This continues until the request is processed or the end of the chain is reached. The detailed steps on how the CoR was implemented are provided below.

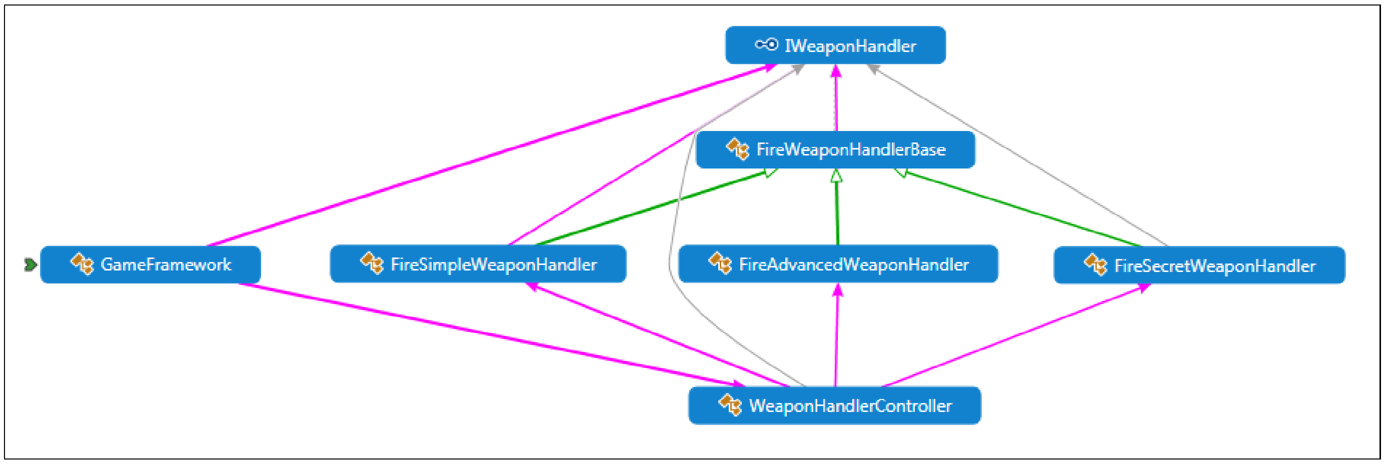

***Figure 1:*** *Class diagram of video game's event handling mechanism utilizing the CoR pattern*

1. **Identify the request (or event) and how it will be represented.**
   In real-world scenarios each handler represents a system. And each system can handle specific requests or requests common to more handlers. For example, a more complex system could require requests with large amounts of data which could increase the complexity level of the implementation. As an alternative, these large objects could be represented by XML strings or identifiers that can be traced to their data stored in a persisted repository. For this application, the request is represented as event arguments intended to be passed between events and their delegates (see ShootWeaponEventArgs class).

2. **Create the base class for the request handler.**
   Common implementations of the CoR pattern call for an abstract base class that is a super class to all request handlers. However, it is generally considered better design to implement to an interface which was what was used by the client to interact with the request handler chain (see IWeaponHandler interface). It should be noted that an abstract class (see FireWeaponHandlerBase class) is still used as the super class for all concrete request handlers. It provides the standard mechanism of chaining successor request handlers while declaring the handler event as an abstract method that child classes should implement to their specific behavior.

3. **Create the concrete request handlers.**
   For the purposes of this example, three concrete request handlers (FireSimpleWeaponHandler, FireAdvancedWeaponHandler, FireSecretWeaponHandler) are created. Each handler has unique implementation in how it handles the request. If it cannot handle the request, it passes the request to the next handler in the chain.

4. **Send requests from the client to the request handler base class.**

The final step in implementing the CoR pattern is to have the client (see GameFramework class) send its requests to the request handler. With the CoR pattern, the client does not need to know the details of the request handlers and that is how it is implemented here. A handler factory (WeaponHandlerController) is used to create/retrieve the actual handlers based on the power level of the Player. At this point, the design allows the client to utilize the request handler without worrying about any future changes in request handling implementation. Any changes will be outside the scope of the client, therefore resulting in a flexible system.

## 1.2 Analysis

The benefits of utilizing the CoR pattern for this application include:

- The **client is decoupled** from the details of the request handler which allows other objects to possibly handle the request as well.
- Request handlers behave similarly to objects designed with the Decorator pattern which allows for request handlers to **change their behavior and responsibilities dynamically**.
- The CoR pattern is useful for applications where the **request handler is not known in advance**.

The liabilities of utilizing the CoR pattern for this application include:

- **Applications can be difficult to debug**. For instance, this application has a bug in the implementation of its request handlers. Each handler only checks for a max value of the power level, but does not check for a lower bound. This could lead to scenarios where a player is not powered-up to use advanced weapons, but based upon how the request handler is chained together, could enter and execute the request handler that is meant for advanced weapons.
- **Unhandled requests** can be realized if none of the request handlers are able to handle the request. This has the possibility of two negative impacts: 1) nothing happens with the request (ignored), and 2) if the chain is lengthy and process intensive, the unhandled request is guaranteed to propagate through each request handler in the chain resulting in increased latency and a degraded user experience in the application.
- A **break in the chain** can be realized if the successor is never called upon in the actual method that handles the request.

Given the pros/cons of the CoR pattern, the utilization of this pattern for the application is still recommended. The biggest reason to support this is that unhandled requests and broken chains can be easily remedied with design/code that has little to no negative impact on the application as a whole. For this application, the positives far outweigh the negatives and achieve a flexible system that is ready for future updates and changes.

## 2. Forwarder-Receiver Pattern

The Forwarder-Receiver (FR) pattern is best utilized for systems requiring communication between the components, or peers. It describes how peer-to-peer (P2P) communications should be implemented by addressing problems that arise when implementing inter-process communications (IPC). The pattern introduces the concept of forwarders and receivers for the purposes of hiding the specifics of the communication mechanisms from the peers. In this way, the peers may act as a client, a server, or both.

### 2.1 Step-by-Step Design

The application that will be used to exercise the capabilities of the FR pattern is a lightweight and simplistic battlefield communications program. Its primary purpose is to allow different battlefield units (soldiers on the ground, battleships off the coast, planes offering air support, etc.) to communicate with one another. Because the application could be deployed on different hosting systems, the application should be independent from a specific IPC mechanism.

Figure 2 illustrates the overall design of the FR pattern used for this application and Appendix B provides the implementation of this design. The details of the underlying IPC mechanism for sending or receiving messages are hidden from the peers by encapsulating all system-specific functionality into separate components. The detailed steps on how the FR was implemented are provided below.

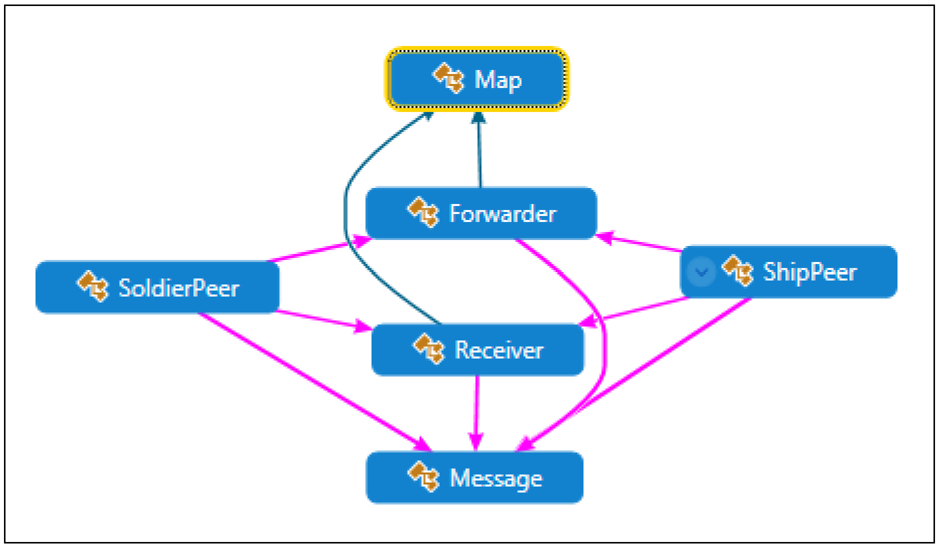

***Figure 2:*** *Class diagram of battlefield communicator's high-level design utilizing the Forwarder-Receiver pattern*

1. **<u>Specify a name-to-address mapping.</u>**
   Considering the simple nature of this application, a basic name space mapping made up of standard strings is used. The strings will serve as hash keys in a hash table of addresses.

2. **<u>Specify the message protocols to be used between peers and forwarders.</u>**
   The Message class defines the message protocol that will be used between peers.

3. **Choose a communication mechanism.**
   The performance impact related to the communication between peers should be minimal and efficient. For this reason, the chosen IPC mechanism for this application is TCP (Transmission Control Protocol). It is a widely used mechanism for network services that most operating systems use today. The forwarders and receivers will use .NET's `TcpClient` and `TcpListener` objects, respectively, to achieve TCP-based communications.

4. **Implement the forwarder.**
   The forwarder component is responsible for marshaling an IPC message and delivering it to the receiver on behalf of the peer. It provides an interface as an abstraction of the TCP communication mechanism (see Forwarder class). When the forwarder sends the message, it specifies the name of its peer so that the remote peer knows who to send the response to.

5. **Implement the receiver.**
   The receiver component is solely responsible for receiving messages that are transmitted by a peer's forwarder. Like the forwarder component, it offers an interface that is an abstraction of a specific IPC mechanism. However, unlike the forwarder, the receiver includes the capabilities for receiving and unmarshaling messages (see Receiver class).

6. **Implement the peers of your application.**
   The peer components (ShipPeer, SoldierPeer) are responsible for application tasks. They may be located in a different process, or different machine. The forwarder component is used to send messages to other peers and the receiver component is used to receive messages from other peers. These messages are either requests (peer A → remote peer B) or responses (peer B → peer A (request originator)).

7. **Implement a start-up configuration.**
   As mentioned in step 1, standard strings will be used as hash keys for addresses composed of an IP address and a port number (see Map class). This is a static class that requires no special configuration. C# console applications like this do provide a mechanism for loading values like this from configuration files, but the Map class serves the same purpose. The one benefit of using a configuration file is that the mappings can be changed without touching of the code.

## 2.2 Analysis

The benefits of utilizing the FR pattern for this application include:

- Requires that **efficient IPC mechanisms** be established.
- Requires that the **IPC mechanisms be encapsulated**. Peers are unaffected by changes to the forwarder/receiver.
- It offers a **general interface for sending and receiving messages and data** across process boundaries.
- **Pre-defined name space mapping**. This is good in that it provides layer of security in that it guarantees that only the mapped components can communicate.

The liabilities of utilizing the FR pattern for this application include:

- **No location transparency**. Clients must be aware of, or dependent on, the name and location of their servers.
- The chosen IPC mechanism can **restrict portability**. Not all systems of required peers may support the IPC mechanism chosen for the FR-based application.
- **Pre-defined name space mapping**. For the very reason that this is a benefit of the FR pattern, it is equally a liability because FR-bases applications are difficult to adapt if the distribution of peers may change at run-time. It all depends on the desired behavior of the application.

Given the pros/cons of the FR pattern, the utilization of this pattern for this application is NOT RECOMMENDED! A key component for an application such as this is the ability to communicate with peers that could change dynamically. As pointed out, one of the main liabilities of this pattern is the inability to support the dynamic reconfiguration of components. It would be ideal to be able to call in air support at critical times, but unless the aircraft is pre-determined and included in the name space map, the application would not be able to provide communications with the aircraft. For this reason, the Forwarder-Receiver pattern is not the best suited for this system. However, the Client-Dispatcher-Server pattern would address these concerns which is acceptable because the FR-based design would only have to be slightly altered to adapt to the Client-Dispatcher-Server pattern.

# Appendix A – *Chain of Responsibility Implementation*

This section provides the source code for the application created for exercising the Chain or Responsibility pattern.

## *GameFramework*

This class is the main application. It is a simple console application that acts as the client in the Chain of Responsibility pattern. It is ultimately responsible for initiating the execution of the first "link" in the chain of request handlers. It does this, however, without knowing the details of the request handlers because it has been decoupled via the CoR pattern.

```
public class GameFramework
{
    private static void Main(string[] args)
    {
        Log.Initialize();
        new GameFramework().Run();
        Console.ReadKey();
        Log.Close();
    }

    private void Run()
    {
        var simpleWeapon = new Weapon();
        simpleWeapon.Shoot += WeaponShoot;

        var player = new Player {Weapon = simpleWeapon};
        FireWeaponHandler = WeaponHandlerController.Create(player.PowerLevel);
        player.FireWeapon();

        // Upgrade player
        player.PowerLevel++;
        FireWeaponHandler = WeaponHandlerController.Create(player.PowerLevel);
        player.FireWeapon();

        // Upgrade player again
        player.PowerLevel++;
        FireWeaponHandler = WeaponHandlerController.Create(player.PowerLevel);
        player.FireWeapon();

        // Downgrade player to simple level
        player.PowerLevel = 1;
        FireWeaponHandler = WeaponHandlerController.Create(player.PowerLevel);
        player.FireWeapon();

        // Upgrade player to unsupported level
        player.PowerLevel = 5;
        FireWeaponHandler = WeaponHandlerController.Create(player.PowerLevel);
        player.FireWeapon();
    }

    IWeaponHandler FireWeaponHandler { get; set; }

    void WeaponShoot(object sender, ShootWeaponEventArgs e)
    {
        FireWeaponHandler.HandleRequest(e);
    }
}
```

### *IWeaponHandler*

This class is the interface that all request handlers in the CoR pattern should implement. This provides a decoupling from any abstract or concrete object while providing a good level of system flexibility for any future change requests.

```
public interface IWeaponHandler
{
    IWeaponHandler Successor { get; set; }

    void HandleRequest(ShootWeaponEventArgs e);
}
```

### *FireWeaponHandlerBase*

This class represents the abstract super class that all concrete request handlers should be derived from. The differences in the concrete request handler classes will be implemented in the `HandleRequest` method.

```
public abstract class FireWeaponHandlerBase : IWeaponHandler
{
    public IWeaponHandler Successor { get; set; }

    protected FireWeaponHandlerBase(IWeaponHandler successor)
    {
        Successor = successor;
    }

    public abstract void HandleRequest(ShootWeaponEventArgs e);
}
```

### *FireSimpleWeaponHandler*

This class is a concrete request handler that handles the firing of simple weapons in the game. It will only handle requests where the `Player`'s power level is 1 or lower. There is also logic to protect against broken chains and/or unhandled requests. If there are broken chains or unhandled requests, the `Player`'s weapon will fire as a simple weapon.

```
public class FireSimpleWeaponHandler : FireWeaponHandlerBase
{
    public FireSimpleWeaponHandler(IWeaponHandler successor) : base(successor) { }

    public override void HandleRequest(ShootWeaponEventArgs e)
    {
        if ((e.Player.PowerLevel < 2) || (Successor == null))
            Console.WriteLine("Firing SIMPLE weapon.");
        else Successor.HandleRequest(e);
    }
}
```

### *FireAdvancedWeaponHandler*

This class is a concrete request handler that handles the firing of advanced weapons in the game. It will only handle requests where the `Player`'s power level is 2 or lower. There is also logic to protect against broken chains and/or unhandled requests. If there are broken chains or unhandled requests, the `Player`'s weapon will fire as an advanced weapon.

```
public class FireAdvancedWeaponHandler : FireWeaponHandlerBase
```

```
{
    public FireAdvancedWeaponHandler(IWeaponHandler successor) : base(successor) { }

    public override void HandleRequest(ShootWeaponEventArgs e)
    {
        if ((e.Player.PowerLevel < 3) || (Successor == null))
            Console.WriteLine("Firing ADVANCED weapon.");
        else Successor.HandleRequest(e);
    }
}
```

### *FireSecretWeaponHandler*

This class is a concrete request handler that handles the firing of secret weapons in the game. It will only handle requests where the `Player`'s power level is 3 or lower. The intent of this request handler is for it to be the last possible way that a weapon can fire. If a `Player`'s power level is 4 or highest, then the request would go unhandled and a message is displayed stating as such (a more realistic implementation would throw an exception instead). The good part about this implementation is that broken links are not possible with this request handler since its successor handler is never called upon. However, the bad is that an unhandled request is not ideally handled by the code.

```
public class FireSecretWeaponHandler : FireWeaponHandlerBase
{
    public FireSecretWeaponHandler(IWeaponHandler successor=null) : base(successor) { }

    public override void HandleRequest(ShootWeaponEventArgs e)
    {
        if (e.Player.PowerLevel < 4)
            Console.WriteLine("Firing SECRET weapon.");
        else
            Console.WriteLine("UH-OH!  This power level is unsupported!!");
    }
}
```

### *WeaponHandlerController*

This class serves as a factory of `IWeaponHandler` objects and is intended for use by clients needing access to the concrete request handler classes. The build criteria passed to the factory is based upon the `Player` object's power level.

```
public static class WeaponHandlerController
{
    public static IWeaponHandler Create(int powerLevel)
    {
        if (powerLevel > 3) return new FireSecretWeaponHandler();

        if (powerLevel > 2)
            return new FireSimpleWeaponHandler(
                new FireAdvancedWeaponHandler(new FireSecretWeaponHandler()));
        if (powerLevel > 1)
            return new FireSimpleWeaponHandler(new FireAdvancedWeaponHandler(null));

        if (powerLevel > 0) return new FireSimpleWeaponHandler(null);

        throw new Exception("ERROR: Invalid power level.");
    }
}
```

*ShootWeaponEventArgs*

This class represents the request that is to be passed to the request handlers in the CoR pattern. A `Player` object could have been specified as the request, but this allows for more dynamic requests to be created per future requirements.

```
public class ShootWeaponEventArgs
{
    public Player Player { get; set; }

    public ShootWeaponEventArgs(Player player)
    {
        Player = player;
    }
}
```

*Weapon*

This class is a supplemental class used by the client. It represents a weapon in the `GameFramework`.

```
public class Weapon
{
    public event EventHandler<ShootWeaponEventArgs> Shoot;

    public void Fire(Player p)
    {
        OnShoot(new ShootWeaponEventArgs(p));
    }

    private void OnShoot(ShootWeaponEventArgs e)
    {
        if (Shoot != null) Shoot(this, e);
    }
}
```

*Player*

This class is a supplemental class used by the client. It represents a player in the `GameFramework`.

```
public class Player
{
    public int PowerLevel { get; set; }

    public Weapon Weapon { get; set; }

    public Player()
    {
        PowerLevel = 1;
    }

    public void FireWeapon()
    {
        Weapon.Fire(this);
    }
}
```

Output:

```
Firing SIMPLE weapon.
Firing ADVANCED weapon.
Firing SECRET weapon.
Firing SIMPLE weapon.
UH-OH!  This power level is unsupported!!
```

# Appendix B – *Forwarder-Receiver Implementation*

This section provides the source code for the application created for exercising the Forwarder-Receiver pattern.

## *ShipPeer*

This class represents a peer that is included in the network of peers for the demonstration project for the FR pattern. It is representing a battleship that is supporting soldiers on land so it performs the role of a server in this instance, but could easily switch roles to a client with requests to other supporting ships. It is a simplistic implementation that simply responds with a generic message as each message is received. However, for the purposes of this project it properly demonstrates the capabilities of P2P communications.

```
public class ShipPeer
{
    private const string PeerName = "Battleship";

    static void Main(string[] args)
    {
        Log.Initialize();

        var forwarder = new Forwarder(PeerName);
        var receiver = new Receiver(PeerName);

        Console.WriteLine("Battleship is online and ready to serve!");

        while (true)
        {
            var message = receiver.ReceiveMessage();

            if (message.Data.Equals("Exit")) break;

            Console.WriteLine("{0}> [{1}]{2}", PeerName, message.Sender, message.Data);

            // Do something for client...

            var dataMsg = new Message(PeerName,
                                      string.Format("{0} has responded!", PeerName));

            forwarder.SendMessage(message.Sender, dataMsg);
        }

        Log.Close();
    }
}
```

## *SoldierPeer*

This class represents a peer that is included in the network of peers for the demonstration project for the FR pattern. It is representing a single/group of soldier(s) that is engaged in warfare on land. In this instance, it performs the role of a client, but could easily switch roles to a server when supporting requests made by other soldiers or ships. It is a simplistic implementation that simply sends a message to a peer based on what is entered at the console. However, for the purposes of this project it properly demonstrates the capabilities of P2P communications.

```
public class SoldierPeer
{
    private const string PeerName = "Boots";

    static void Main(string[] args)
    {
        Log.Initialize();

        var forwarder = new Forwarder(PeerName);
        var receiver = new Receiver(PeerName);

        do
        {
            Console.Write("{0}> ", PeerName);
            var msg = Console.ReadLine();

            if (string.IsNullOrWhiteSpace(msg))
            {
                forwarder.SendMessage("Battleship", new Message(PeerName, "Exit"));
                break;
            }

            var dataMsg = new Message(PeerName, msg);

            forwarder.SendMessage("Battleship", dataMsg);

            var response = receiver.ReceiveMessage();

            Console.WriteLine("{0}> [{1}]{2}", PeerName, response.Sender, response.Data);

            // Do something with response...

        } while (true);

        Log.Close();
    }
}
```

## *Map*

This class serves as the de-facto run-time configuration of the P2P components. All components included in this map can communicate with the other through the application.

```
public static class Map
{
    public static Dictionary<string, string> Clients =
        new Dictionary<string, string>
        {
            {"Battleship", "192.168.137.107:8989"},
            {"Boots", "192.168.137.107:9090"},
        };
}
```

## *Forwarder*

This class is the message forwarder in the FR pattern. Its primary responsibilities include marshaling the message data and sending the message based on the specific IPC mechanism. All of these details are hidden from the peers. To send a message to a remote peer, the peer invokes the `SendMessage` method of its `Forwarder` object, passing a `Message` object (according to the messaging protocol) as an argument. The `SendMessage` method must convert messages to a format that the underlying IPC mechanism understands. For this purpose, it calls the `Marshal` method to properly format the message, and then calls `SendMessage` to transmit the IPC message data to a remote receiver. The `Marshal` method creates a byte array where the first four bytes are the size of the message and the remaining bytes are composed of the sender's name and the pertinent data.

```
public class Forwarder
{
    private string PeerName { get; set; }

    public Forwarder(string peerName)
    {
        PeerName = peerName;
    }

    public void SendMessage(string destination, Message msg)
    {
        try
        {
            var id = Map.Clients[destination].Split(':');
            var ip = id[0];
            var port = Int32.Parse(id[1]);

            var data = Marshal(msg);

            // Create a TcpClient.
            // Note, for this client to work a TcpServer must be connected to the same
            // address as specified by the server's IP and port number combination.
            var client = new TcpClient();
            client.Connect(ip, port);

            // Get a client stream for reading and writing.
            var stream = client.GetStream();

            // Send the message to the connected TcpServer.
            stream.Write(data, 0, data.Length);

            // Close everything.
            stream.Dispose();
            client.Close();
        }
        catch (ArgumentNullException e)
        {
            Console.WriteLine("ArgumentNullException: {0}", e);
        }
        catch (SocketException e)
        {
            Console.WriteLine("SocketException: {0}", e);
        }
    }

    private byte[] Marshal(Message msg)
    {
        // Translate the passed message into ASCII and store it as a Byte array.
        var data = Encoding.ASCII.GetBytes(string.Format("[{0}]{1}",
                                                    msg.Sender, msg.Data));
        var numBytes = 4 + data.Length;
        var packet = new byte[numBytes];

        Buffer.BlockCopy(BitConverter.GetBytes(numBytes), 0, packet, 0, 4);
        Buffer.BlockCopy(data, 0, packet, 4, data.Length);

        return packet;
    }
}
```

*Receiver*

This class is the message receiver in the FR pattern. Its primary responsibilities include receiving the message (based on the specific IPC mechanism) and unmarshaling the message data. All of these details are hidden from the peers. When the peer wants to receive a message from a remote peer, it invokes the `ReceiveMessage` method of its `Receiver` object, and a `Message` object containing the received data is returned. `ReceiveMessage` invokes `Receive`, which uses the functionality of the underlying IPC mechanism to receive IPC messages. After message reception `ReceiveMessage` calls `Unmarshal` to convert IPC messages to a format that the peer understands. For this application, the `Unmarshal` method ignores the first four bytes of the message, but this could be used to verify the success or failure of a transmitted message by comparing the size of the message received with the value received in those first four bytes.

```
public class Receiver
{
    public string PeerName { get; private set; }

    public Receiver(string peerName)
    {
        PeerName = peerName;
    }

    private Message Unmarshal(byte[] buffer)
    {
        var msgSize = BitConverter.ToInt32(buffer, 0) - 4;

        var msg = Encoding.ASCII.GetString(buffer, 4, msgSize);
        var sender = msg.Substring(1, msg.IndexOf(']') - 1);
        var senderLength = sender.Length + 2;
        var data = msg.Substring(senderLength, msg.Length - senderLength);

        return new Message(sender, data);
    }

    private byte[] Receive()
    {
        byte[] bytes = null;
        TcpListener server = null;

        try
        {
            var id = Map.Clients[PeerName].Split(':');
            var ip = id[0];
            var port = Int32.Parse(id[1]);

            // Set the TcpListener on the port specified in the namespace map.
            var localAddr = IPAddress.Parse(ip);
            server = new TcpListener(localAddr, port);

            // Start listening for client requests.
            server.Start();

            // Buffer for reading data
            String data = null;

            Console.WriteLine("{0} : ", PeerName);

            // Perform a blocking call to accept requests.
            TcpClient client = server.AcceptTcpClient();

            // Get a stream object for reading and writing
            NetworkStream stream = client.GetStream();

            // Read in the first 256 bytes of the network stream.
            bytes = new byte[256];
            stream.Read(bytes, 0, bytes.Length);

            // Shutdown and end connection
            client.Close();
        }
        catch (SocketException e)
        {
            Console.WriteLine("SocketException: {0}", e);
        }
        finally
        {
            // Stop listening for new clients.
            if (server != null) server.Stop();
        }
        return bytes;
    }

    public Message ReceiveMessage()
    {
        return Unmarshal(Receive());
    }
}
```

## *Message*

This class defines the message protocol that will be used by peers to send messages back and forth between peers. The data for this application will be standard strings, but a more realistic implementation could include more complex objects according to an application's needs.

```
public class Message
{
    public string Sender { get; private set; }

    public string Data { get; private set; }

    private Message() { }

    public Message(string sender, string data)
    {
        Sender = sender;
        Data = data;
    }
}
```

<u>Output:</u>

| SoldierPeer (client) | ShipPeer (server) |
|---|---|
| `Boots> This is a test exercise.` | `Battleship is online and ready to serve!` |
| `Boots :` | `Battleship :` |
| `Boots> [Battleship]Battleship has responded!` | `Battleship> [Boots]This is a test exercise.` |
| `Boots> Battleship, please help!!` | `Battleship :` |
| `Boots :` | `Battleship> [Boots]Battleship, please help!!` |
| `Boots> [Battleship]Battleship has responded!` | `Battleship :` |
| `Boots> Thank you Navy!!` | `Battleship> [Boots]Thank you Navy!!` |
| `Boots :` | `Battleship :` |
| `Boots> [Battleship]Battleship has responded!` | |
| `Boots>` | |

## Non-Direct Activity Report

| Date | Duration (minutes) | Specific Task / Activity |
|---|---|---|
| 24-May-2015 | 80 | Research for project #3. |
| 6-Jun-2015 | 112 | Research for project #3. |
| 13-Jun-2015 | 312 | Research for project #3. |
| 14-Jun-2015 | 615 | Research for project #3. |
| 15-Jun-2015 | 324 | Research for project #3. |
| 16-Jun-2015 | 168 | Research for project #3. |
| 17-Jun-2015 | 230 | Research for project #3. |
| 18-Jun-2015 | 86 | Research for project #3. |
| **Sum for Report #1** | **1570** | **/ 900 (1 week @ 900/wk)** |
| **Sum for Report #2** | **1991** | **/ 900 (1 week @ 900/wk)** |
| **Sum for Report #3** | **1927** | **/ 900 (1 week @ 900/wk)** |
| **Sum for Report #4** | | **/ 1800 (2 weeks @ 900/wk)** |
| **Sum for Class** | **5488** | **/ 4500 (5 weeks @ 900/wk)** |