Structural Patterns
Structural patterns are concerned with how classes and objects are composed to form larger structures.

Decorator Pattern
Decorator pattern attaches additional behavior without inheritance.

```
namespace DecoratorPattern
{

    interface IEmail
    {
        string Write(string message);
    }

    class Email : IEmail
    {
        public string Write(string message)
        {
            return message;
        }
    }

    class EmailWithAddressing : IEmail
    {
        IEmail email;
        string recipient;

        public EmailWithAddressing(IEmail email, string recipient = null)
        {
            this.email = email;
            this.recipient = recipient;
        }

        public string Write(string message)
        {
            string resultMessage = email.Write(message);
            if (String.IsNullOrEmpty(recipient))
            {
                resultMessage = "Dear Sir or Madam,\n" + resultMessage;
            } else
            {
                resultMessage = String.Format("Dear {0},\n", recipient) + resultMessage;
            }
            return resultMessage;
        }
    }

    class EmailWithSigning : IEmail
    {
        IEmail email;

        public EmailWithSigning(IEmail email)
        {
            this.email = email;
        }

        public string Write(string message)
        {
            string resultMessage = email.Write(message);
            resultMessage += "\nYours sincerely";
            return resultMessage;
        }
    }


    class EmailWithAttachment : IEmail
    {
        IEmail email;

        public EmailWithAttachment(IEmail email)
        {
            this.email = email;
        }

        public string Write(string message)
        {
            return email.Write(message);
        }

        public string AddAttachment(string attachment)
        {
            return "Added attachment: " + attachment;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Decorator Pattern\n");
            IEmail email = new Email();

            Console.WriteLine("Basic Component:");
            Console.WriteLine(email.Write("How are you?"));
            Console.WriteLine();

            EmailWithAddressing addressedEmail = new EmailWithAddressing(email, "Bob");
            EmailWithSigning signedEmail = new EmailWithSigning(addressedEmail);
            EmailWithAttachment emailWithAttachment = new EmailWithAttachment(signedEmail);
            Console.WriteLine("Decorated Component:");
            Console.WriteLine(emailWithAttachment.Write("How are you?"));
            Console.WriteLine(emailWithAttachment.AddAttachment("Image"));

            Console.ReadLine();
        }
    }

}
```

Proxy Pattern

Proxy pattern controls the creation of and access to others objects.
There are several kinds of proxies:

- Virtual proxies. Hands creation of an object over to another object.
- Authentication proxies. Checks the access permissions for a request.
- Remote proxies. Sends requests across the network.
- Smart proxies. Changes requests before sending them on.

```
namespace ProxyPattern
{
    class HelpDeskSystem
    {

        private class HelpDesk
        {
            static SortedList<string, HelpDesk> users = new SortedList<string, HelpDesk>();
            string user;

            public static bool IsUnique(string userName)
            {
                return !users.ContainsKey(userName);
            }

            internal HelpDesk(string userName)
            {
                user = userName;
                users[userName] = this;
            }

            internal void CreateTicket(string message)
            {
                Console.WriteLine(String.Format("{0} created new ticket: {1}", user, message));
            }

        }

        public class HelpDeskProxy
        {
            HelpDesk myHelpDesk;
            string user;
            string password;
            bool loggedIn = false;

            void Register()
            {
                Console.WriteLine("\nLet's register your HelpDesk account");
                do
                {
                    Console.WriteLine("All HelpDesk accounts must be unique");
                    Console.Write("Type in a user name: ");
                    user = Console.ReadLine();
                } while (!HelpDesk.IsUnique(user));
                Console.Write("Type in a password: ");
                password = Console.ReadLine();
                Console.WriteLine("Thanks for registering with HelpDesk");
            }

            bool Authenticate()
            {
                Console.Write(String.Format("Welcome {0}. Please type in your password: ", user));
                string supplied = Console.ReadLine();
                if (password == supplied)
                {
                    loggedIn = true;
                    Console.WriteLine("Logged into HelpDesk");
                    if (myHelpDesk == null)
                    {
                        myHelpDesk = new HelpDesk(user);
                    }
                    return true;
                }
                Console.WriteLine("Incorrect password");
                return false;
            }

            void Check()
            {
                if (!loggedIn)
                {
                    if (password == null)
                    {
                        Register();
                    }
                    if (myHelpDesk == null)
                    {
                        Authenticate();
                    }
                }
            }

            public void CreateTicket(string message)
            {
                Check();
                if (loggedIn)
                {
                    myHelpDesk.CreateTicket(message);
                }
            }
        }

    }

    class Program : HelpDeskSystem
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Proxy Pattern");

            HelpDeskProxy userA = new HelpDeskProxy();
            userA.CreateTicket("My printer doesn't work...");

            HelpDeskProxy userB = new HelpDeskProxy();
            userB.CreateTicket("I can't access my email...");

            Console.ReadLine();
        }
    }
}
```

Bridge Pattern

Bridge pattern separates an abstraction from its implementation. It is useful when a new version of software should replace an existing version, but the older version must still run for its existing client base.

```
namespace BridgePattern
{
    // Bridge
    interface IBridge
    {
        void CreateTicket(string message);
    }

    // Abstraction
    class Portal
    {
        IBridge bridge;

        public Portal(IBridge aHelpDesk)
        {
            bridge = aHelpDesk;
        }

        public void CreateTicket(string message)
        {
            bridge.CreateTicket(message);
        }

    }

    class HelpDeskSystem
    {
        // The Subject
        private class HelpDesk {...}

        // The Proxy
        public class HelpDeskProxy {...}

        // New version of Proxy
        public class OpenHelpDeskProxy : IBridge
        {
            HelpDesk myOpenHelpDesk;
            static int usersCount;
            string user;

            public OpenHelpDeskProxy(string userName)
            {
                user = userName;
                usersCount++;
                myOpenHelpDesk = new HelpDesk(user + "-" + usersCount);
            }

            public void CreateTicket(string message)
            {
                myOpenHelpDesk.CreateTicket(message);
            }
        }
    }
    class Program : HelpDeskSystem
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Bridge Pattern");

            HelpDeskProxy userA = new HelpDeskProxy();
            userA.CreateTicket("I forgot my password.");

            Console.WriteLine();
            string userName = "userB";
            Portal userB = new Portal(new OpenHelpDeskProxy(userName));
            userB.CreateTicket("Internet connection is too slow.");

            Console.ReadLine();
        }
    }
}
```

Composite Pattern

Composite pattern arranges structured hierarchies so that single components and groups of components can be treated in the same way. Pattern has to deal with two types: Components and Composites of those components. Both types should implement common operations of the same interface.

```
namespace CompositePattern
{
    public interface IComponent<T>
    {
        T Name { get; set; }
        void Add(IComponent<T> c);
        IComponent<T> Find(T p);
        string Output(int depth = 0);
    }

    public class Component<T> : IComponent<T>
    {
        public T Name { get; set; }

        public Component(T name)
        {
            Name = name;
        }

        public void Add(IComponent<T> c)
        {
            Console.WriteLine("An item already exists");
        }

        public IComponent<T> Find(T p)
        {
            if (Name.Equals(p))
                return this;
            else
                return null;
        }
        public string Output(int depth = 0)
        {
            return new String(' ', depth) + Name.ToString() + "\n";
        }
    }

    class Composite<T> : IComponent<T>
    {
        public T Name { get; set; }
        List<IComponent<T>> list;

        public Composite(T name)
        {
            Name = name;
            list = new List<IComponent<T>>();
        }

        public void Add(IComponent<T> c)
        {
            list.Add(c);
        }

        // Recursively looks for an item
        public IComponent<T> Find(T p)
        {
            if (Name.Equals(p)) {
                return this;
            }
            IComponent<T> found = null;
            foreach (IComponent<T> component in list)
            {
                found = component.Find(p);
                if (found != null)
                {
                    break;
                }
            }
            return found;
        }

        // Returns items in a format indicating their level in the composite structure
        public string Output(int depth = 0)
        {
            StringBuilder sb = new StringBuilder(new String(' ', depth));
            sb.Append(Name.ToString() + " (" + list.Count() + ")\n");
            foreach (IComponent<T> component in list)
            {
                sb.Append(component.Output(depth + 2));
            }
            return sb.ToString();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Create structure of directories and files
            IComponent<string> logDir = new Composite<string>("log");
            for (int i = 1; i < 7; i++)
            {
                logDir.Add(new Component<string>(i + ".log"));
            }
            IComponent<string> backupDir = new Composite<string>("backup");
            for (char c = 'a'; c <= 'i'; c++)
            {
                backupDir.Add(new Component<string>(c + ".bck"));
            }
            IComponent<string> rootDir = new Composite<string>("/");
            rootDir.Add(logDir);
            rootDir.Add(backupDir);
            rootDir.Add(new Component<string>("readme.txt"));

            // Output structure
            Console.WriteLine(rootDir.Output());

            // Find an item in structure and output it
            Console.WriteLine(rootDir.Find("5.log").Output());

            Console.ReadLine();
        }
    }
}
```

Flyweight Pattern

Flyweight pattern promotes an efficient way to share common information present in small objects that occur in a system in large numbers.

```
namespace FlyweightPattern
{
    public interface IFlyweight
    {
        void Load(string filename);
        void Display(PaintEventArgs e, int row, int col);
    }

    public struct Flyweight : IFlyweight
    {
        Image pThumbnail;

        public void Load(string filename)
        {
            pThumbnail = new Bitmap("../../pictures/" + filename + ".jpg").
                GetThumbnailImage(100, 76, null, new IntPtr());
        }

        public void Display(PaintEventArgs e, int row, int col)
        {
            e.Graphics.DrawImage(pThumbnail, col * 105 + 5, row * 120 + 40,
                pThumbnail.Width, pThumbnail.Height);
        }
    }

    public class FlyweightFactory
    {
        Dictionary<string, IFlyweight> flyweights = new Dictionary<string, IFlyweight>();

        public FlyweightFactory()
        {
            flyweights.Clear();
        }

        public IFlyweight this[string index]
        {
            get
            {
                if (!flyweights.ContainsKey(index))
                {
                    flyweights[index] = new Flyweight();
                }
                return flyweights[index];
            }
        }
    }

    class Program
    {
        static FlyweightFactory album = new FlyweightFactory();
        static Dictionary<string, List<string>> groups = new Dictionary<string, List<string>>();

        public void LoadGroups()
        {
            var myGroups = new[]
            {
                new
                {
                    Name = "Animals", Members = new [] { "Jellyfish", "Koala", "Penguins" }
                },
                new
                {
                    Name = "Favorite", Members = new [] { "Koala", "Penguins", "Tulips", "Lighthouse" }
                }
            };

            foreach (var g in myGroups)
            {
                groups.Add(g.Name, new List<string>());
                foreach (string filename in g.Members)
                {
                    groups[g.Name].Add(filename);
                    album[filename].Load(filename);
                }
            }
        }

        public void DisplayGroups(Object source, PaintEventArgs e)
        {
            int row = 0;
            foreach (string groupName in groups.Keys)
            {
                int col = 0;
                e.Graphics.DrawString(groupName, new Font("Arial", 12),
                    new SolidBrush(Color.Black), new Point(0, row * 120 + 10)
                );
                foreach (string filename in groups[groupName])
                {
                    album[filename].Display(e, row, col);
                    col++;
                }
                row++;
            }
        }

        public class Window : Form
        {
            public Window()
            {
                this.Height = 550;
                this.Width = 600;
                this.Text = "Picture Groups";
                Program program = new Program();
                program.LoadGroups();
                this.Paint += new PaintEventHandler(program.DisplayGroups);
            }
        }

        static void Main(string[] args)
        {
            Application.Run(new Window());
        }
    }

}
```

Adapter Pattern

Adapter Pattern is useful wherever there is code to be wrapped up and redirected to a different implementation. Adapter can put in varying amounts of work to adapt base implementation to new interface. The simplest adaption is just to reroute a method call to one of a different name

```
namespace AdapterPattern
{
    // Existing way of implementation
    public class Adaptee
    {
        public double Divide(double a, double b)
        {
            return a / b;
        }
    }

    // Required standard
    public interface ITarget
    {
        string DivideRound(int a, int b);
    }

    // Implementing new standard via existed implementation
    public class Adapter : Adaptee, ITarget
    {
        public string DivideRound(int a, int b)
        {
            return "Round " + (int)System.Math.Round(Divide(a, b));
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Before the new standard, precise value: ");
            Adaptee first = new Adaptee();
            Console.WriteLine(first.Divide(5, 3));

            Console.Write("Moving to the new standard: ");
            ITarget second = new Adapter();
            Console.WriteLine(second.DivideRound(5, 3));

            Console.ReadLine();

        }
    }

}
```