Composite Design Pattern in action

```
class Program
{
    static void Main(string[] args)
    {
        var developer = new IndividualContributor("Alice", "Developer");
        var designer = new IndividualContributor("Bob", "Designer");

        var manager = new Manager("Charlie", "Manager");
        manager.Add(developer);
        manager.Add(designer);

        var ceo = new Manager("Diana", "CEO");
        ceo.Add(manager);

        ceo.DisplayDetails();
    }
}
```

The Composite Design Pattern is a structural design pattern that allows you to compose objects into tree structures to represent part-whole hierarchies. This pattern lets clients treat individual objects and compositions of objects uniformly. Let's dive into the details of this pattern, its components, and a real-world example to understand its implementation and usage.

@lijotech

Components of the Composite Design Pattern

- **Component**: This is an abstract class or interface that defines the common operations for both simple and complex objects of the tree structure.

- **Leaf**: This represents the individual objects in the composition. A leaf doesn't have any children.

- **Composite**: This represents the complex objects that may have children. A composite can contain both leaf and other composite objects.

@lijotech

Component

The **Employee** class is the abstract component that defines the common operations for both individual contributors and managers.

```
public abstract class Employee
{
    protected string name;
    protected string position;

    public Employee(string name, string position)
    {
        this.name = name;
        this.position = position;
    }

    public abstract void DisplayDetails();
    public abstract int GetTotalSubordinates();
}
```

@lijotech

Leaf

The **IndividualContributor** class represents the individual employees who do not have any subordinates.

```
public class IndividualContributor : Employee
{
    public IndividualContributor(string name, string position) :
base(name, position) { }

    public override void DisplayDetails()
    {
        Console.WriteLine($"{position}: {name}");
    }

    public override int GetTotalSubordinates()
    {
        return 0; // Individual contributors have no subordinates
    }
}
```

@lijotech

Composite

The **Manager** class represents the managers who can have subordinates. This class implements the **IEmployeeOperations** interface to manage the subordinates

```
public interface IEmployeeOperations
{
    void Add(Employee employee);
    void Remove(Employee employee);
}
public class Manager : Employee, IEmployeeOperations
{
    private List<Employee> _subordinates = new List<Employee>();
    public Manager(string name, string position) : base(name, position) { }
    public void Add(Employee employee)
    {
        _subordinates.Add(employee);
    }
    public void Remove(Employee employee)
    {
        _subordinates.Remove(employee);
    }
    public override void DisplayDetails()
    {
        Console.WriteLine($"{position}: {name} (Total subordinates: {GetTotalSubordinates()})");
        foreach (var subordinate in _subordinates)
        {
            subordinate.DisplayDetails();
        }
    }
    public override int GetTotalSubordinates()
    {
        int totalSubordinates = _subordinates.Count;
        foreach (var subordinate in _subordinates)
        {
            totalSubordinates += subordinate.GetTotalSubordinates();
        }
        return totalSubordinates;
    }
}
```

@lijotech

Usage Scenarios

- **Company Organizational Structures**: Representing employees and managers.

- **File System Hierarchies**: Files and directories can be treated uniformly.

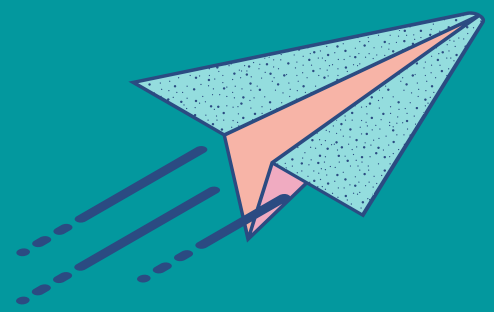
- **UI Components**: Widgets that contain other widgets.

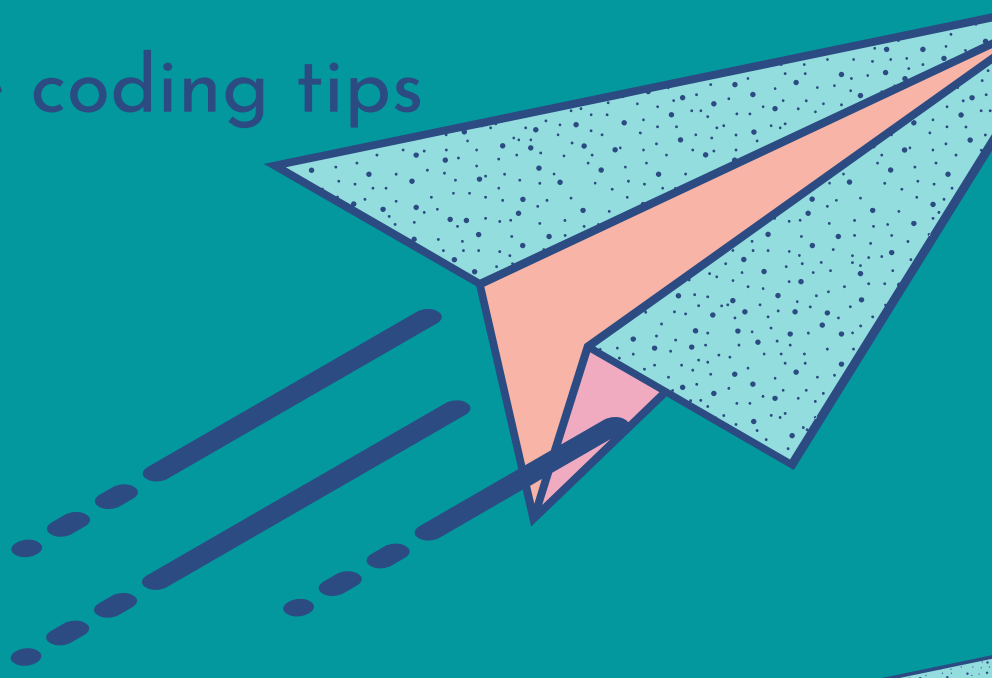
- **Graphics Drawing Applications**: Shapes that can contain other shapes.

@lijotech

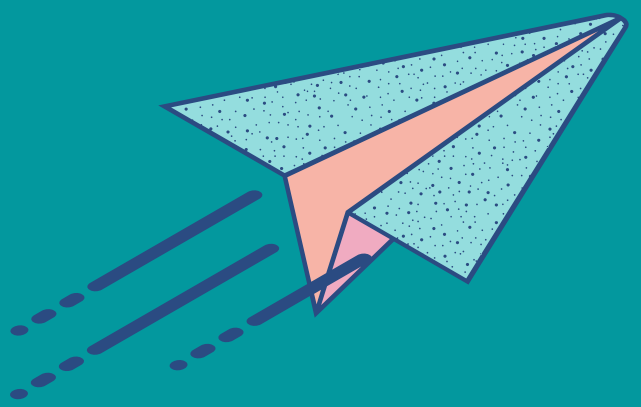
Stay Focused & Happy Coding

Follow Lijo Sebastian for more coding tips





@lijotech



@lijotech